On Adapation via Ultrastability

Candiate Number: 52804

Contents

1 Introduction

'The biologist must view the brain ... like every other organ in the body, as a specialized means to survival' - William Ross Ashby.

In Ashby's seminal work 'A Design for a Brain' [2] he states two fundamental questions that must be considered when examining how adaptive behaviour arises. Firstly we must 'identify the nature of the change which shows as learning' and secondly, 'find out why such changes should tend to cause better adaptation for the whole organism'.

Ashby's thesis is that adaptation occurs through mechanisms of change and stability. His underlying argument is that all organisms have certain properties, which he termed essential variables, that must be kept within certain physiological conditions. These essential variables are often closely linked dynamically, so that changes in one lead to changes in others.

Survival occurs under Ashby's thesis when a line of behaviour moves no essential variables outside of their limits. Adaptation can therefore be viewed as the maintenance of internal stability; the homeostatic regulation of the essential variables to ensure survival. If stability cannot be maintained then the system must change and search for a new set of conditions that can maintain stability. If the system can find such a point, then it can be considered to have adapted.

Ashby proposes a powerful conceptual framework for developing agents without reliance on embedded teleological control. Instead of providing organisms with purposeful goal seeking behaviours, he showed that 'purposeless' mechanistic models could provide rich mechanisms to generate self-induced adaptive behaviour through the maintenance of internal stability. Such systems are termed 'Ultrastable'.

Recent work in neuroscience has shown homeostatic regulatory mechanisms [17] [18] control synaptic efficacy in the brain. These mechanisms are driven by the need to be adaptive to activity-dependent change while maintaining internal stability.

In one of the few applications of Ashby's Ultrastability thesis, Di Paolo [5] applied homeostatic regulation of synaptic activity to the evolution of agents engaging in phototaxis. In a very interesting result, he showed that the agents could adapt to sensory inversion, even though they were not evolved specifically for this task. This result echoed the work of Kohler [13] and Held [10].

The tentative reason given by Di Paolo for adaptation is that the methodology implicitly links stability and the desired behaviour during the evolutionary process. Di Paolo theorises that the model creates a stable attractor for patterns of sensorimotor activities but acknowledges that there are many open questions relating to the mechanism that drive adaptation in his model.

In recreating this work, I aim to investigate the nature of these changes to gain further understanding of how Ultrastability leads to adaptation and specifically to investigate Di Paolo's claim about the behavioural linkage.

1.1 A Design for a Brain

In 1952, William Ross Ashby proposed a conceptual framework to investigate how adaptation and specifically somatic adaptation, may arise in living systems [2]. Ashby's focus was on showing how the brain could be regarded as highly mechanistic but still show adaptive behaviour. By considering organisms as dynamical systems, representing them as a state determined systems, an organism's behaviour can therefore be explained because '... its physical and chemical nature at that moment allowed no other option'.

Ashby's highly mechanistic model describes how an organism's behaviour changes by studying how the variables representing that organism and its observable behaviour change in time. Central to this approach is that an organism can only be described by considering the agent and its environment as a whole, because the organism affects the environment and the environment the organism. 'The organism and the environment

form a whole and must be view as such'.

By coupling the agent and its environment to form a state-determined system, an environment is particular to that organism, because the environment is defined as a system whose variables affect and are affected by the organism. Each organism experiences a different environment.

Establishing a dynamic systems viewpoint of behaviour, Ashby's thesis continues that adaptation is the conservation of stability, and especially the stability of what he termed 'essential variables'. Such variables could include, for example, internal temperature, energy or oxygen level in the blood.

Behaviour is said to be adaptive in this framework if it maintains essential variables within their viable limits and ideally at their optimal values. In Ashby's view all adaptive systems are stable systems requiring mechanisms for the homeostatic regulation of essential variables.

Such an adaptive behaviour arises in a two stage process. First, the organism goes from not having an adaptive mechanism to having one and second, this developed mechanism shifts from inactivity to activity. If essential variables cannot exceed set bounds, it therefore presupposes a mechanism for change. Ashby proposed the concept of Ultrastability to account for this change.

1.2 Ultrastability

Ultrastability is defined as the ability to find a stable state for the essential variables under a variety of environmental conditions. Ashby's view was that all adaptive organisms have some regulatory mechanisms constructed via an output to the environment and at least two feedback loops. In the primary loop, the embedded organism interacts with environment via sensorimotor feedback. A second feedback loop exists, often running at a slower speed, from the environment to the essential variables. These essential variables can in turn affect sub-systems causing a change in the organism.

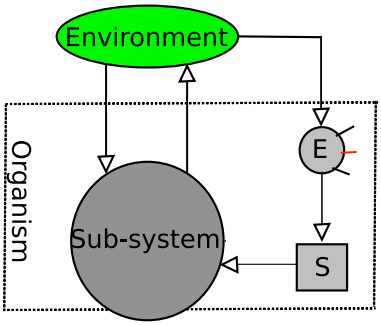


Figure 1: **Ultrastable interaction**

An abstracted figure of an Ultrastable systems is shown in Figure 1. Two systems of variables, a sub-system (some reacting part of the organism) and an environment interact, causing the agent to both modify and be modified by the environment. The arrows to and from the sub-system represent the sensorimotor loop and represents the primary feedback loop in the system. S represents those parameters that affect the sub-system, which have an immediate effect on the subsystem's behaviour (but not on the environment). E represents the essential variables as a dial with limit marks representing their boundaries. These essential variables can have an impact on the S parameters.

Change in an Ultrastable system occurs at two levels. The first type of change is small perturbations to the essential variables caused by the agent-environment coupling. The second type of changes occurs when the essential variables approach or exceed their bounds. A larger perturbation occurs as the essential variables induce a change in S. Thus forcing the organism to find a new equilibrium point by changes in its behaviour.

If the agent can find such an equilibrium, it can be viewed as having adapted.

In Ashby's framework the search for a new equilibrium is induced through step functions that move the current state of the system to a new state. When the essential variables go out of bounds, they cause S to change value via this step mechanism to a different point in state space, thus inducing the reacting part to exhibit modified behaviour. While the choice of step functions here is not interesting biologically, it demonstrates that even random searching, the simplest mechanism for discovering new equilibrium points, is capable of driving adaptive changes. Adaptation in an Ultrastability model shows that multiple feedback loops are required, sensorimotor as well as q secondary mechanism.

The notion of Ultrastability can be expanded to Multistable systems; systems comprised of multiple ultrastable subsystems which are stable to a myriad of perturbations. Ashby demonstrated how such systems lead to the 'dispersion of behaviour' and showed the effectiveness of a multistable system through the construction of the Homeostat. This machine comprised of four randomly coupled ultrastable electro-magnetic systems that would search for new stable equilibriums when perturbed by randomising their connections. Even faced with serious perturbations to its inputs, the system would reach stable state. In a multistable system because the response to disturbances is distributed across the sub-system, the stability of the system as a whole is harder to compromise since no one disturbance can perturb the system as a whole.

1.3 Synaptic Efficacy Regulation

In a summary of recent findings in neuroscience, Turrigiano [17] highlighted some of the biological underpinnings of homeostatic plasticity in neuronal networks. Neural circuitry is driven by the need to change but also the need for functional stability. Activity-dependent change supports refinement of synaptic efficacy through experience to create cortical maps, hypercolums, columns and selectivity, allowing information to be stored [16].

Hebbian changes have long been recognised as one mechanism for driving change in neural circuitry. But without some regulatory components Hebbian changes will destabilise neuronal activity by creating a bimodal distribution of synaptic strengths by strengthening correlated firing pattens and weakening uncorrelated firing patterns. Mechanisms for homeostatic regulation must therefore exist. Mechanisms have been discovered to control intrinsic neuron excitability, synaptic strength and synaptic stabilisation [12].

The firing rate of a neuron must not be too high or too low because networks must remain sensitive to their inputs in order to develop selectivity and store information. A mechanism for modulating the intrinsic excitability of neurons has been seen in the stomatogastric ganglion (STG) of crustaceans [17]. The intracellular concentration level of CA^2+, a calcium ion, is seen to be well correlated with activity. When activity falls, the levels of CA^2+ also falls which modifies the conductance of the neuron. The change in conductance causes output current fall but inward current rises, raising both the excitability of the neuron and the intracellular level of CA^2+.

A second relevant regulatory mechanism for synaptic efficacy is cortical pyramidal neurons. Self-modulation occurs on the incoming synapse strengths adjusting self-activity to control efficacy in the reverse of Hebbian learning. As firing rates increase then excitatory connections are scaled down and vice versa. One aspect is that the time frame this occurs in is much slower than Hebbian learning, creating a double feedback loop with differing timescales, which has strong similarities Ashby's Ultrastable system.

1.4 A minimal model of Ultrastability

Recent work by Beer [3] and others has championed the use of simplier idealized models of adaptation to understand from a dynamically systems viewpoint how adaptive behaviour may occur. Like Ashby, the emphasis is on studying how the dynamics of an agent and its environment are capable of producing effective behaviour.

Di Paolo's study *Homeostatic adaptation to inversion of the visual field and other sensorimotor disruptions* [5] follows in this lineage by studying a mimimal model of Ultrastability influence by Turrigiano's analysis [17]. A minimal agent, guided by homeostasis is shown to engender adaptation to sensory inversion. Plasticity is triggered when the level of neural activity becomes too high or too low, inducing the system to search for new stable configurations, thus adapting to restore stability. Di Paolo's agents show clearly how plasticity effects the generation of behaviour, and behaviour affects plasticity, leading to the creation of invariants and habits.

The principle issues addressed by recreating this work are Ashby's two fundamental questions. First, to identify the changes in such a system that lead to learning and second, to explain why such changes should lead to adaptation for the organism as a whole.

Di Paolo gives a tentative explanation of how adapatation arises in this model. While the agents are not evolved to adapt to inversion, it is suggested that evolution has created a linkage between structural stability and the desired behaviour. This link occurs because the process shapes 'the space of weight change so that there is a stable attractor when a certain pattern of sensorimotor activity is present'. [7]

In restaging Di Paolo's study I hope is to understand how minimal Ultrastable systems produce adaptation and what the exact nature of this adaptation is. Once the nature of of the mechanisms driving adaptation are understood then it's hoped that an explanation will arise as to why only half of the agent's selected for sensor inversion could adapt in Di Paolo's experiment.

2 Model

In this study, a population of minimal agents are evolved to exhibit phototaxsis on a series of light sources. Each agent is evaluated in a flat, infinite two-dimensional environment which has a single light source at any time. Agents are evaluated on a series of light sources and during each trial the agents are encouraged to remain close to the light sources.

Light sources are placed at a distance of $\in [10 : 25]$ times the agent's radius from the agent, for a period of time T_s. T_s is chosen from the interval [0.75T, 1.25T] where T is either 400 or 800 (representing 2000 or 4000 time steps, respectively).

OpenGL [1] is used to display the agents, their paths and the light sources - (Figure 2).

2.1 Agent

Each agent is modelled as a simple circular agent (Figure 3) with two light sensors and two diametrically opposed motors which can drive the agent around. Each motor is driven by a single motor neuron, whose firing rate is scaled to set the motors speed.

The agent's body is defined as having a small size and very small mass, allowing the motor output to be the 'tangential velocity at the point of the body where the motor is located' [5]. The translational movement of the whole robot is calculated using the velocity of its centre of mass (1) and the rotational movement by calculating the angular speed, ω (2), where V_l and V_r are the motors' velocities and r is the agent's radius.

$$V_c = \frac{V_r + V_l}{2} \tag{1}$$

$$\omega = \frac{V_r - V_l}{2r} \tag{2}$$

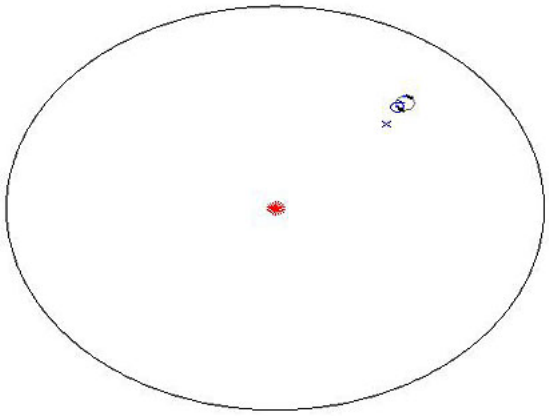


Figure 2: **Screen shot of agent and environment**. Showing a rendered agent towards a light source (red star). The circle around the light source is a measure of its intensity. The small circle on the agent represents that its left sensor is active.

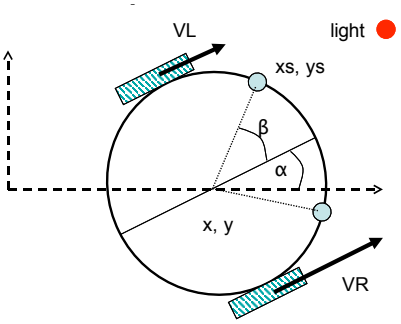


Figure 3: **Simple Low Inertia Wheeled Robot** [14]

The new position and angle of the agent at each time step is given by (3-5), where x and y are the current position of the agent, θ is it's orientation and δt the size of the integration timestep.

$$x(t+1) = x(t) + V_c.cos(\theta).\delta t \tag{3}$$
$$y(t+1) = y(t) + V_c.sin(\theta).\delta t \tag{4}$$
$$\theta(t+1) = \theta(t) + \omega.\delta t \tag{5}$$

For the experiments in this paper, the body's radius is set at 4.0 [6].

2.1.1 Sensors

Each agent has two light sensors, separated by 120°, symmetrically placed at the front of the agent. Light from the sensors is can be occluded by the agent's body and each sensor has a viewing cone of 80°.

At each time step, the angle from the agent to the light source is first calculated and then adjusted by the agent's rotation. This angle is then used to determine if the agent's sensors are active. If the sensors are active, the sensor's positions are calculated using the following formula (6) and (7).

$$x_s = x + r.cos(\alpha + \beta) \tag{6}$$

$$y_s = y + r.sin(\alpha + \beta) \tag{7}$$

The distance from the sensor to the light, D_s can then be computed and light impinges on the sensor with an intensity, I, inversely proportional to the square of the distance to the light source. (8).

$$I = \frac{\text{Source Intensity}}{D_s^2} \tag{8}$$

2.1.2 The Controller

A 4-neuron fully connected Continuous Time Recurrent Neural Network (CTRNN) is used as the agent's controller. Each neuron is a CTRNN node is describedlo by equations (9) and (10). Each neuron's state is controlled by y_i, representing the cell's potential, τ_i is the decay constant, bi the bias, z_j the firing rate of the jth neuron, w_{ij} the strength of synaptic connection from node i to node j, and I_i input from external sources (the sensors).

$$\tau_i \dot{y_i} = -y_i + \sum_j w_{ji} z_j + I_i \tag{9}$$

The firing rate of a the jth neuron is defined by the standard logistic function (10), in which y_j is the potential of the neuron and b_j its bias. The firing rate is scaled to the range [0,1] by this function.

$$z_j = \frac{1}{1 + exp^{-(y_j + bj)}} \tag{10}$$

There are two sensory neurons and two motor neurons, nominally defined to be the first two and last two nodes in the CTRNN respectively.

2.2 Controller Plasticity

The agents' controllers can be evolved to regulate synaptic activity. Plastic changes in the CTRNN occur locally on incoming synapses when a neuron's firing rate is either too high or too low, introducing 'a potentially ultrastable element in each neuron' [7].

The plasticity of each connection is governed by both the synaptic activity and a plasticity rule encoded genetically. Homeostasis is not evolved for, but becomes linked via evolution with the goal of having stable phototatic behaviour in the long term.

The plasticity rules are given in equations 11-14, where Δw_{ij} is the change per unit of time to a particular synaptic weight, w_{ij} and z_i and z_j are the firing rates of the presynaptic and postsynaptic neurons respectively.

The plasticity rules are built around four parameters: (n_{ij}), the learning rate, (z^o_{ij}), a threshold value, (p_j), the degree of local plastic facilitation and (δ) a linear damping factor that constrains change within allowed weight values.

R0: Bounded Hebbian learning.

$$\Delta w_{ij} = \ \delta \ n_{ij} \ p_j \ z_i \ z_j \tag{11}$$

R1: Dampen potentiation or depression of the presynaptic neuron when the synaptic efficicy is too high or too low.

$$\Delta w_{ij} = \ \delta \ n_{ij} \ p_j \ (z_i - z^0_{ij}) z_j) \tag{12}$$

R2: Dampen potentiation or depression of the postsynaptic neuron when the synaptic efficicy is too high or too low.

$$\Delta w_{ij} = \ \delta \ n_{ij} \ p_j \ z_i (z_j - z^o_{ij}) \tag{13}$$

R3: No plasticity rule:

$$\Delta w_{ij} = 0 \tag{14}$$

p_j, the local plasticity parameter is governed by the plasticity function shown in Figure 4. No plastic changes occur when the firing rate is in bounds, approximately [0.119, 0.881]. If the firing rate increases in strength to take it out of bounds, the local plasticity parameter increases linearly facilitating plastic changes, until p_j reaches its maximum value of 1. When firing rates fall out of bounds the inverse happens, p_j decreases linearly until it reaches its minimum value of -1. The actual change depends on the sign of n_{ij} and so each neuron can act independently to facilitate plastic change in a given direction.

δ, the linear damping function ensures that weight values are kept in bounds and its form is shown in Figure 5. The threshold function, z^o_{ij} depends linearly on the current weight value; also shown in Figure 5.

Weights are updated every integration step using the following equation (15).

$$w_{ij}(t+1) = w(t) + \Delta w_{ij} \tag{15}$$

2.3 Genetic Algorithm

The genotype of each agent is divided into a real component and an integer component. The real component has 42[1] genetic components comprising a motor and sensor gain for the network, a time and bias value for each node and a synaptic weight and learning rate for every connection in the network. Left/right symmetry is enforced through having shared sensor and motor gains for the agent.

The integer component has 16 loci with 4 possible values, representing the learning rule for plasticity change associated with every connection in the network.

[1]For an 4 node CTRNN.

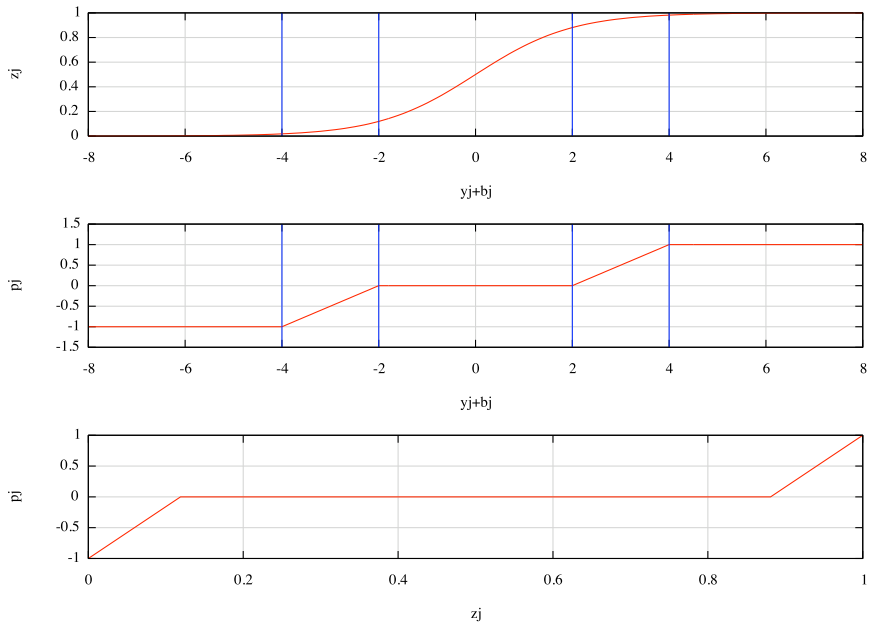
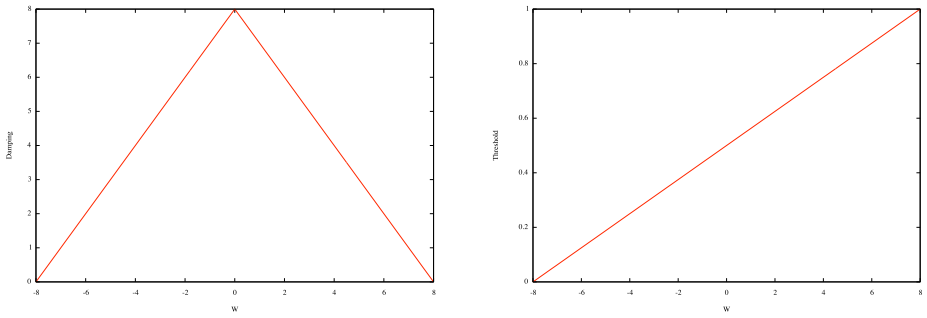


Figure 4: **Plasticity as a function of 'cell potential'.** Top: Firing rate. Middle: Degree and direction of plastic changes in response to neural activity. Bottom: Plastic facilitation as function of firing rate.



(a) Linear Damping Function (b) Threshold Function

Figure 5: **Damping and Thresholding functions for Plasticity**

All the real values are linearly scaled to be in the range [0,1] apart from the sensor and motor gains which are scaled exponentially. All real values are mutated via a creep mutation operation which adds a small random vector to the genetic value during every mutation [6]. Mutations that fallout side of the range [0,1] are cropped.

The integer components of the genotype are mutated via point mutation and selected from the range $[0, 3]$.

2.3.1 Microbial GA

A microbial tournament genetic algorithm (GA) [9] is used to select and test agents against a sequence of five lights. Two agents are selected at random to compete against other in every step of the GA. Agents' fitness against the light sequence is assessed by placing them into the simulated arena. At the beginning of each trial, the agent is placed in the arena without a light source to enable the CTRNN to stabilise.

The fitness scores of both agents are assessed and the selected agents bred, using crossover and mutation, to form a new offspring, which then replaces the lowest scoring agent. A mutation probability of 0.5 is used for all real components and a probability of 0.1 is used for integer components. The probability of recombination is 0.5 for both real and integer genotype sections.

The GA is repeated until no perceivable fitness increase is observed.

2.3.2 Evaluating fitness

The agent's behaviour is assessed using a tri-modal fitness function (16), where F_d is a measure of how near to the source an agent is, F_p is a measure of time spent near a source and F_h is a measure of the homeostatic behaviour of an agent's neurons. The results of all three sub-fitness functions are ranged $\in [0, 1]$.

The constants α, β and γ are weightings to adjust the contribution of each sub-fitness function and always sum to 1. Typical values are $\alpha = 0.2$, $\beta = 0.64$ and $\gamma = 0.16$ respectively, which favours activity (and thus phototaxis) near the lightsources.

$$F = \alpha F_d + \beta F_p + \gamma F_h \tag{16}$$

F_d is computed by measuring the reduction in final and initial starting positions for the agent ($F_d = (1 - D_f)/D_i$, where D_f is the final distance to the source and D_i is the initial distance. If $D_f > D_i$ then F_d is set to 0.

F_p is the proportion of time over the evaluation period that the agent is within $4 * radius$ units of the light source. F_h is the time-averaged proportion of neurons that act homeostatically, without inducing plasticity in the controller.

3 Results

The following section presents some early results for three evolved agent populations[2]. Non-plastic CTRNNs are first evolved as a control, then a population with plastic rules but no reward for homeostatic behaviour and finally a group of agents who are rewarded both for phototaxis and internal stability.

[2]Time constraints prevented analysis of much of the data recorded during agent evaluation. The data is presented here to demonstrate the dynamics of each agent type.

3.1 Phototaxis without plasticity

To gain an understanding of how agents evolve to perform photoaxis, the agent population was constrained to use the R0 rule only (no plasticity) and with the fitness function modifiers set to $\alpha = 0.2, \beta = 0.8, \gamma = 0$. This population forms the control population against which agents with plastic rules can be compared.

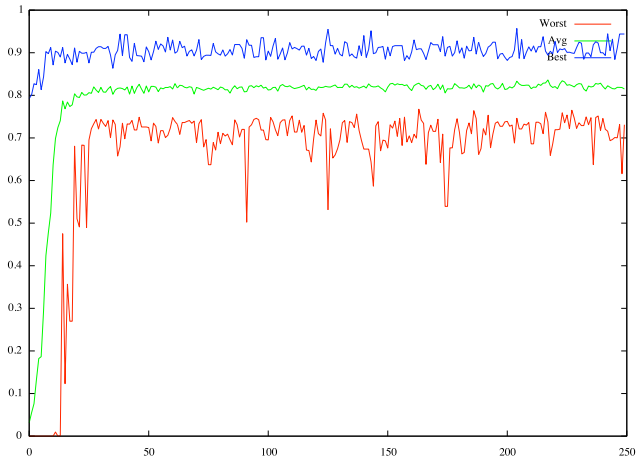


Figure 6: **Fitness of population during evolution per epoch**. Each epoch consists of 60 tournaments.

Figure 6 shows the worst, best and average fitness scores for the population during evolution. The fittest agent at the completion of the GA, after 12,000 tournaments, was then used to test the long-term stability of phototactic behaviour in a non-plastic agent.

Figure 7 shows the CTRNN controller evolved for this agent. The interesting photatic technique employed by this agent is to rotate on the spot sweeping the sensors over the light source. The agent keeps rotating until its 'back' is to the light source and then reverses rapidly towards the light. After some distance it begins to rotate again and the searching is repeated until the agent reaches the source.

Stable solutions for phototaxis appear quickly in the population and Figure 8 shows that behaviour is stable in long term (plots are truncated for ease of display). Each spike in distance represents the replacement of a light source. The assigned fitness values during this run are also given in this figure.

Figure 9 and Figure 10 show the firing rate, z_i, of each neuron and the cell potentials, y_i, during the presentation of the first few light sources (approximately 4500 timesteps). Figure 10 shows the cell potentiations during this evaluation.

3.2 Phototaxis with plasticity

Plastic changes are are now allowed for all agents in the population but there is no reward for homeostatic behaviour in the fitness function ($\gamma = 0$). The fitness of the evolving populations has slightly different dynamics to the control group. Figure 11 shows the worst, best and average fitness scores for the population during evolution. The rate of convergence towards good solutions is markedly slower in the average and worst cases.

As before, the fittest agent at the completetion of the GA, is used to test the long term stability of phototactic behaviour. This agent's CTRNN is given by Figure 12.

Figure 13 shows that behaviour is stable in long term (plots are truncated for ease of display). The agent

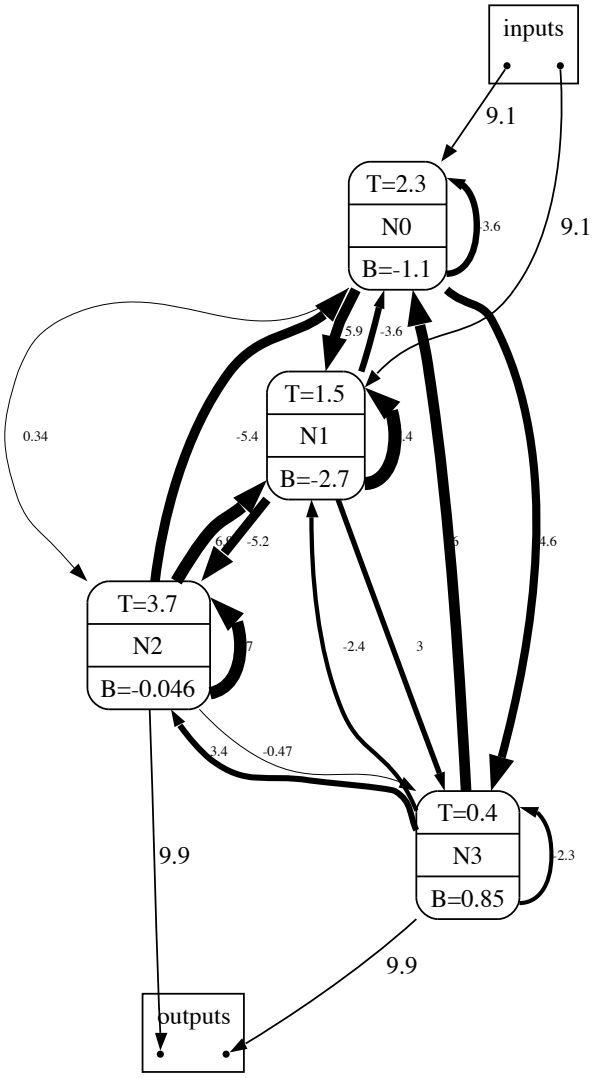
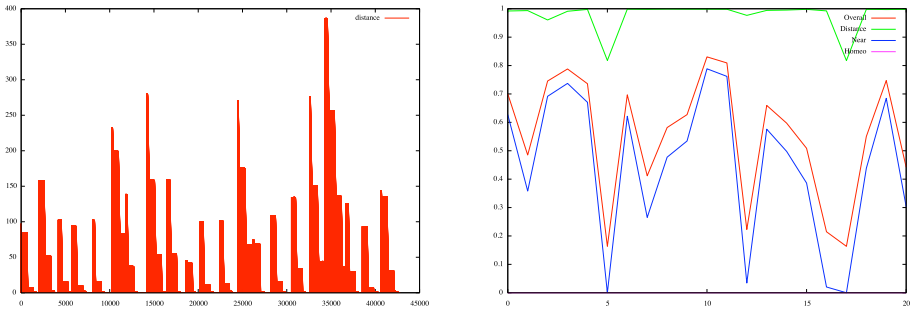


Figure 7: **Evolved non-plastic CTRNN controller.** Nodes are shown with T=τ and bias values. Line stroke weight captures the weight of the connection between the nodes. As only R0 rules are seen all connections are drawn in black.



(a) Distance to source (b) Internal Fitness

Figure 8: Stability of solution over presentation of lights

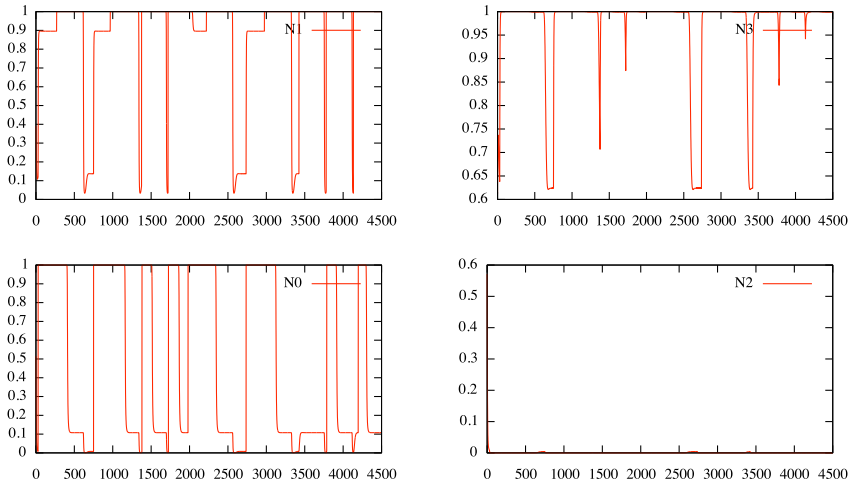


Figure 9: **Firing rates for each neuron in the non-plastic CTRNN.** N0 is in the bottom left and neurons are presented in a clockwise direction.

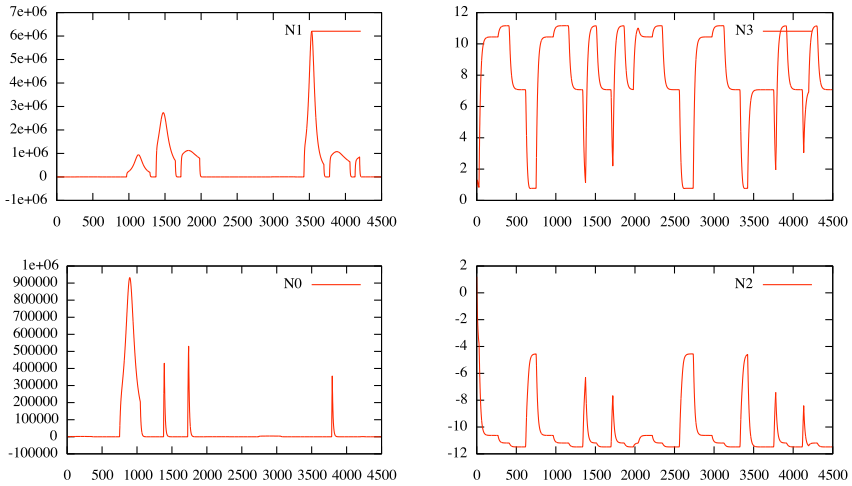


Figure 10: **Cell potentials for each neuron in the non-plastic CTRNN.** N0 is in the bottom left and neurons are presented in a clockwise direction.

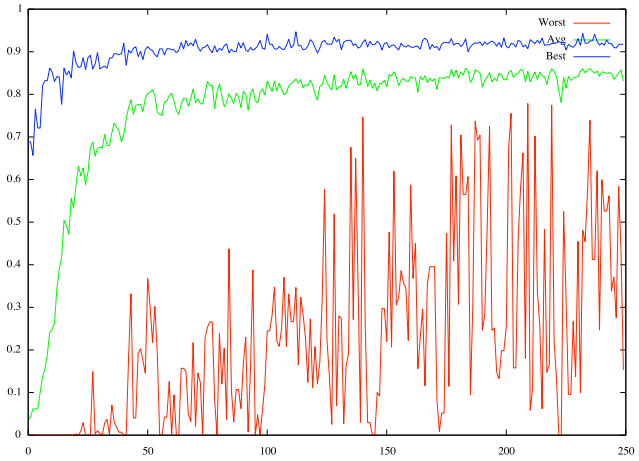


Figure 11: **Fitness of population during evolution per epoch**. Each epoch consists of 60 tournaments.

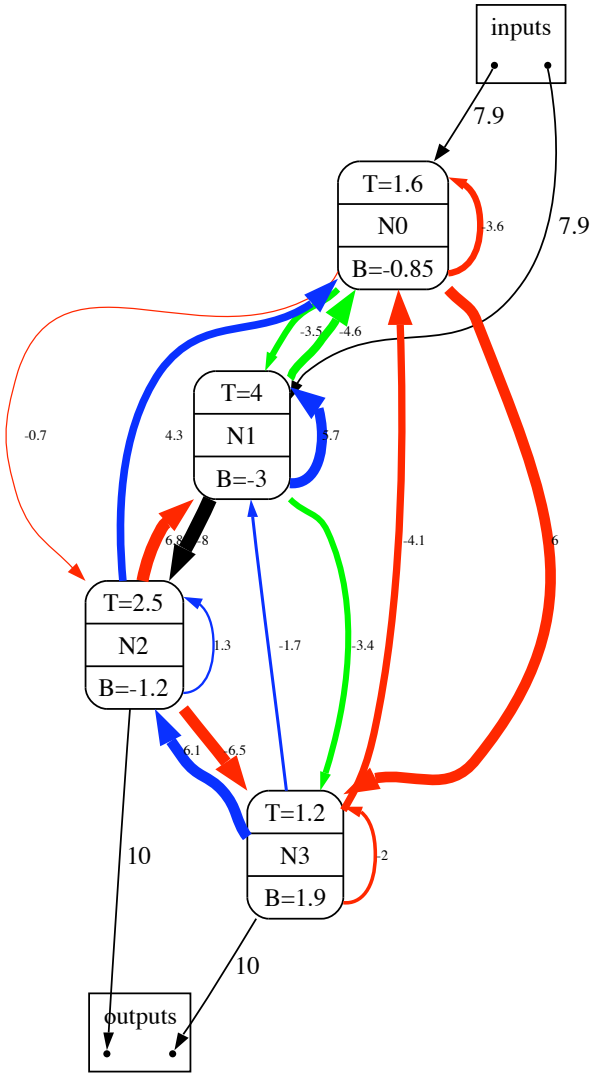
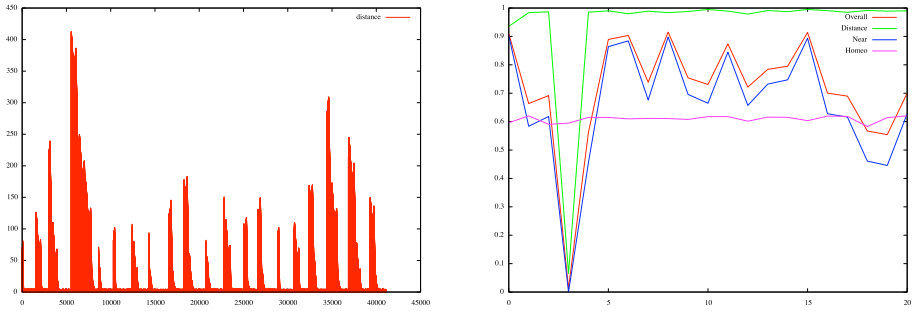


Figure 12: **Evolved plastic CTRNN controller.** The coloured arrows represent the following learning rules: R0 (blue), R1 (red) and R2 (green).



(a) Distance to source (b) Internal Fitness

Figure 13: Stability of solution over presentation of lights for a plastic controller

shows a behaviour seen commonly in non-plastic agents of moving in a series of arcs rather than straight lines.

Figure 14 and Figure 15 show the firing rate, z_i, of each neuron and the cell potentials, y_i during the presentation of the first few light sources (approximately 4500 timesteps). Figure 10 shows the cell potentiations during this evaluation.

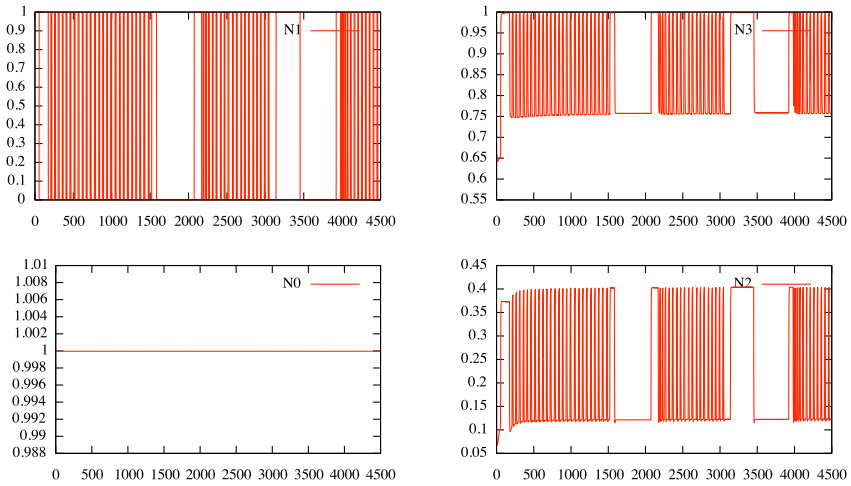


Figure 14: **Firing rates for each neuron in the plastic CTRNN.** N0 is in the bottom left and neurons are presented in a clockwise direction.

3.3 Phototaxis with plasticity and stability

The final population group is evolved with the fitness function rewarding stability (fitness function modifiers are $\alpha = 0.21, \beta = 0.64, \gamma = 0.15$).

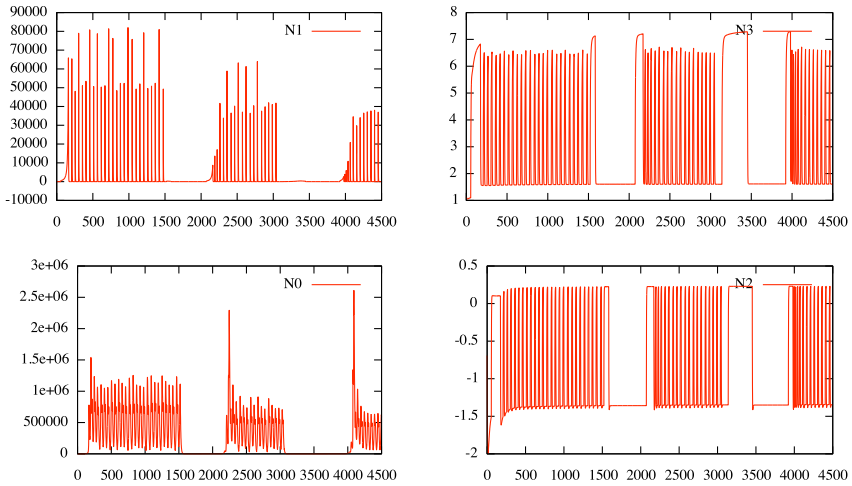


Figure 15: **Cell potentials for each neuron in the plastic CTRNN.** N0 is in the bottom left and neurons are presented in a clockwise direction.

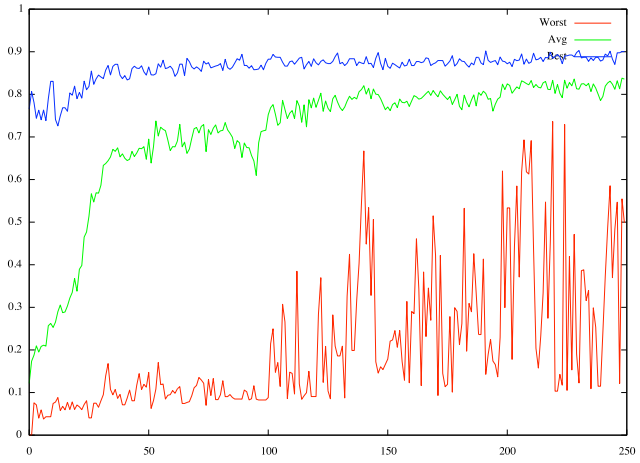


Figure 16: **Fitness of population during evolution per epoch**. Each epoch consists of 60 tournaments.

As before, the fittest agent at the completetion of the GA, is used to test the long term stability of phototactic behaviour. This agent's CTRNN is given by Figure 17.

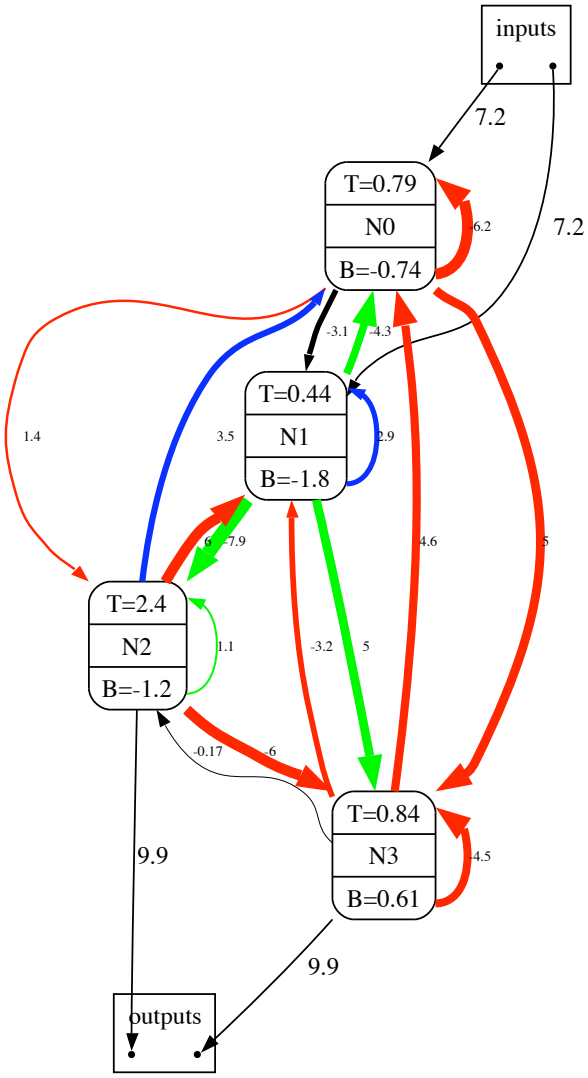
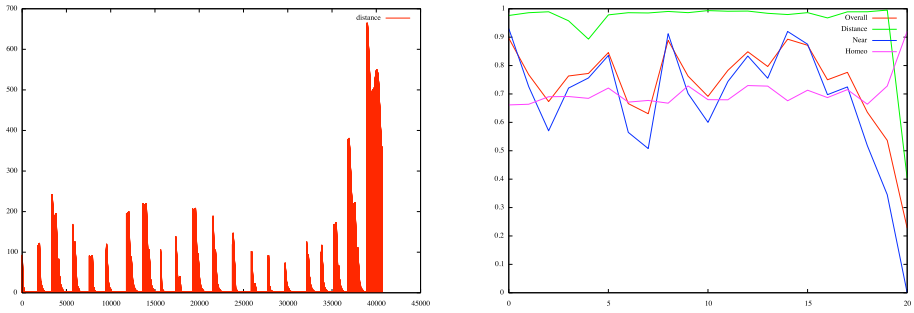


Figure 17: **Evolved plastic & homeostatic CTRNN controller.** The coloured arrows represent the following learning rules: R0 (blue), R1 (red) and R2 (green).

Figure 18 shows that behaviour is reasonably stable in long term (plots are truncated for ease of display). The agent shows a behaviour seen commonly in non-plastic agents of moving in a series of arcs rather than straight lines. It's movement seems the least 'purposeful' of all the agents suggesting the evolution is still shaping the solution.

Figure 14 and Figure 19 show the firing rate, z_i, of each neuron and the cell potentials, y_i during the presentation of the first few light sources (approximately 4500 timesteps). Figure 20 shows the cell potentiations during this evaluation.



(a) Distance to source (b) Internal Fitness

Figure 18: Stability of solution over presentation of lights for a plastic controller

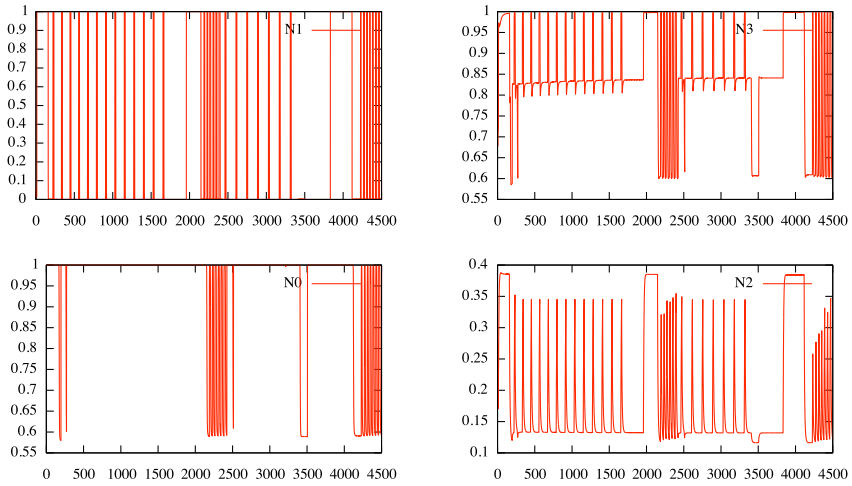


Figure 19: **Firing rates for each neuron in the plastic & homeostatic CTRNN.** N0 is in the bottom left and neurons are presented in a clockwise direction.

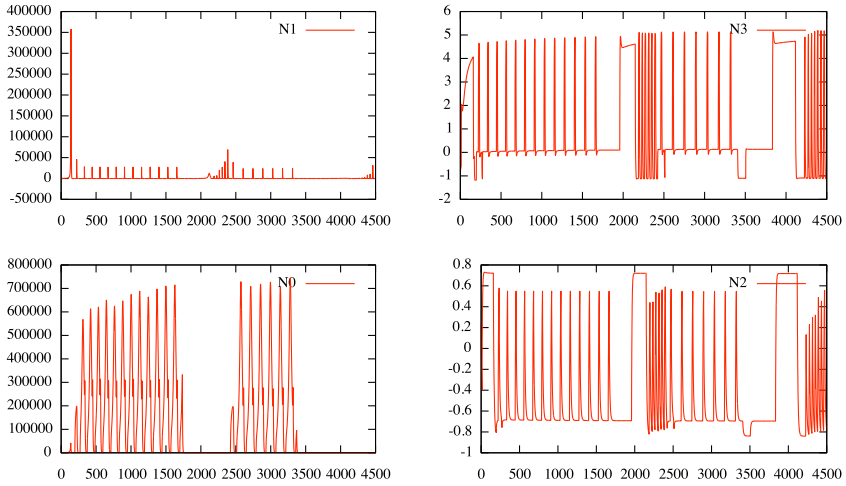
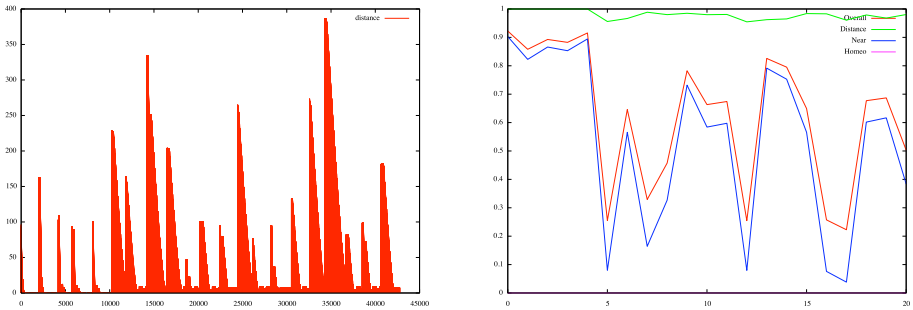


Figure 20: **Cell potentials for each neuron in the plastic & homeostatic CTRNN.** N0 is in the bottom left and neurons are presented in a clockwise direction.

3.4 Response to sensory inversion

Once long term stability of phototaxis had been evolved in the three agent populations, the highest scoring agents were subjected to sensory inversion at a fixed point during a trial with twenty light sources. After the presentation of the fifth light source, the connections to the input neurons are swapped.

Unfortunately the control population of non-plastic agents 'adapted' perfectly without requiring any internal change. The rotate and then move behaviour was replaced with a mechanism of direct frontal approach that had a better overall performance. Looking at the distance and fitness graph for this agent, Figure 21, during the presentation of a light source after inversion the agent linearly closes the distance.

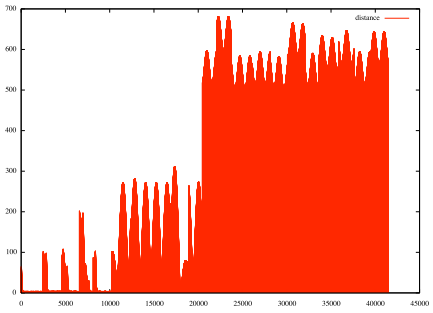


(a) Distance to source

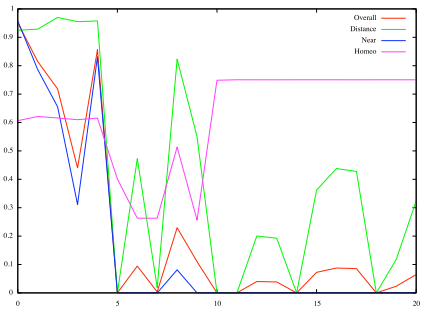
(b) Internal Fitness

Figure 21: **Stability of solution during sensory inversion for a non plastic agent.** Sensory inversion happens after the fifth light.

The plastic controller without reward for homeostasis cannot adapt to sensory inversion, Figure 22. Its performance looks better on the distance plot as it moved in a series of circles in the arena that coincided with proximity to sensor locations. Once the sensors were repositioned outside of this circling area performance dropped markedly.



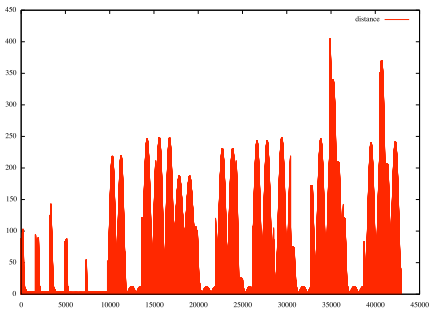
(a) Distance to source



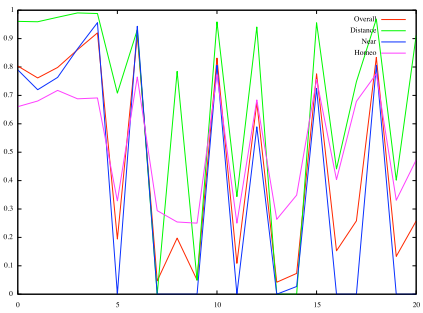
(b) Internal Fitness

Figure 22: **Instability of solution during sensory inversion for a plastic-agent.** Sensory inversion happens after the fifth light.

The plastic & homeostatic controller recovers somewhat from sensory inversion - Figure 23. Its success seems to depend on getting input from both sensors and the 'recovered' performance is partial.



(a) Distance to source



(b) Internal Fitness

Figure 23: **Partial stability of solution during sensory inversion for a plastic & homeostatic agent.** Sensory inversion happens after the fifth light.

4 Conclusions

This work further confirms that the Ultrastability hypothesis of Ashby can be used to develop controllers for minimal agents. In this report it has been shown that agents evolved for both internal stability and phototaxis are able to adapt to sensory inversion, whereas plastic agents that evolved without reward for internal stability could not. Surprisingly non-plastic agents could also adapt to inversion. Much further work is required to analyse the data now being produced by the simulation, but one possible explanation is that the agent's strategy does not rely on using the difference between the light sensor values.

Unfortunately, the preliminary results for this recreation have limited impact on explaining how Ultrastability leads to adaptation. In lieu of concrete evidence, the results of [5] and [19] will be discussed instead.

When sensorimotor disruption occured, the homoeostatic regulation of firing rates forced the agents to search for new stable behaviours. In his paper, Di Paolo provided a tentative argument as to why only agents that were rewarded for maintaining internal stability could adapt to sensorimotor perturbations. He states that the evolutionary process links implicit stability and phototaxis and thus shapes the weight space creating a 'stable attractor when a certain weight pattern of sensorimotor activity is present'.

As noted in Di Paolo's paper such an explanation is only speculative in the absence of further evidence because the rate of weight changes are slow during the initial period after sensor inversion.

There is also unreliability in the creation of this attractor as only half of selected agents could adapt to sensory inversion successfully. This unreliability indicates that there may be many other possible outcomes. There could be other stable attractors in the weight space creating bifurcations in behaviour. Rather than an artefact created by evolution, the linking between behaviours may not arise and instead the outcome is created because the behaviours do not interfere with each other, given certain circumstances.

A secondary criticism of the explanation, noted by Di Paolo, is that weight changes occur for a long period before adaptive behaviour emerges. The results produced in this recreation currently provide little insight into what triggers the onset of adaptation. Ashby's original thesis requires multiple feedback loops working on different temporal scales. The primary feedback loop is the fast sensorimotor loop and the secondary, the mechanism that triggers plasticity, is slower. It may be that adaptation is triggered by appearance of specific sensorimotor patterns during the search for new equilibriums.

There are many further questions to be explored in the application of the Ultrastability thesis [19]. As noted by Williams, one drawback of selecting for stability is that the evolutionary process takes longer and average fitness levels are lower for homeostatic agents. Agents evolved for stability cannot take advantage of saturation regions to provide cheap solutions to the phototaxis task.

Plasticity in this study is, linked to only single nodes via genetically codified rules. 'Individual cells behave homeostatically by facilitating local plasticity whenever their activity goes out of bounds' [5]. The rules used are unstable, driving the connections weights to the extremes [7] and while being biologically inspired lack biological plausibility [12]. Synaptic efficacy has been shown to be regulated by Spike Timing Dependent Plasticity (STDP) *in vitro* [15]. Recent work on STDP [12] [18] provides plausible, stable and competitive mechanisms to regulate plasticity (without limiting it to single nodes) as well as providing mechanisms for homeostasis through the balancing long-term depression and potentiation, overall neural firing, action potential levels amongst other parameters.

While in later work, Di Paolo examined STDP as mechanism for control [7], but outside of the framework of Ultrastable systems. STPD regulatory mechanisms have been shown to drive the formation of temporal neural structures [16] through activity and thus shaping the response of an organism to sensorimotor patterns. It is the author's view that these mechanism for synaptic efficacy change create strong synergies with Ashby's framework.

Further work is also required to study multi-stable systems for robot control, in which the ultrastable sub-systems respond to differing environmental perturbations and diffuse essential variables. By introducing the requirement for stability in multiple essential variables, by the nature of the mechanistic mechanism, purposefulness is embedded in the robot without teleological control.

It is plausible to argue that in Ultra or Multi-stable systems teleology isn't avoided, because the selection of essential variables and their maintenance of stability is purpose imposed externally. However, as survival is the maintenance of some form of separation of the environment, such a minimal framework seems to be a requisite. Behaviour then emerges through the survival of the organism and that behaviour is dependent on the unique environment associated with each organism and is linked to the plasticity in the organism required to adapt to change in that environment.

Therefore, it's the author's firm belief that further study of Ashby's thesis is essential to understand how the fundamental mechanisms underlying organistic behaviour are shaped by the need to change and the need to stay the same.

References

[1] Angel, E. (1997) *Interactive computer graphics: a top-down approach with OpenGL.* Addison-Wesley.

[2] Ashby, W.R. (1960). *Design for a Brain: the origin of adaptive behaviour.* Chapman and Hall, 2nd Edition, 1960.

[3] Beer, R.D. (1996) *Toward the evolution of dynamical neural networks for minimally cognitive behavior.* Proc. of the Fourth Intl. Conf. on Simulation of Adaptive Behavior, pp. 421-429.

[4] Buckley, C., Bullock, S. & Cohen, N. (2005) *Timescale and stability in adaptive behaviour.* Proc. of the Eighth European Conference on Artificial Life, pp. 292-301.

[5] Di Paolo, E. A. (2000). *Homeostatic adaptation to inversion of the visual field and other sensorimotor disruptions.* From Animals to Animats, Proc. of the Sixth Intl. Conf. on the Simulation of Adaptive Behavior, pp. 440-449.

[6] Di Paolo, E. A. (2000). *Behavioral coordination, structural congruence and entrainment in a simulation of acoustically coupled agents.* Adaptive Behavior Vol. 8(1), pp. 25 - 46.

[7] Di Paolo, E. A. (2003) *Evolving spike-timing dependent plasticity for single-trial learning in robots.* Philosophical Transactions of the Royal Society A. Vol. 361, pp. 2299 - 2319.

[8] Di Paolo, E.A. (2003) *Organismically-inspired robotics: homeostatic adaptation and teleology beyond the closed sensorimotor loop.* In Dynamical Systems Approach to Embodiment and Sociality, Advanced Knowledge International, pp.19-42.

[9] Harvey, I. (2001) *Artificial evolution: A continuing SAGA.* Proc. of 8th Intl. Symposium on Evolutionary Robotics (ER2001).

[10] Held, R. (1965) *Plasticity in sensory-motor systems*, Scientific American, Vol. 213(5), pp. 84-94.

[11] Ikegami, T. & Suzuki, K. (2007) *From homeostatic to homeodynamic self.* In Proc. Modelling Autonomy Workshop, San Sebastiáan.

[12] Kepecs, A., van Rossum, M.C.W, Song, S. and Tegner, J. (2002) *Spike-timing-dependent plasticity: common themes and divergent vistas.* Biological Cybernetics. Vol. 87, p. 446458.

[13] Kohler, I. (1962) *Experiments with goggles.* Scientific American, Vol. 206, pp. 62-72.

[14] Seth, A. *A simple, low inertia wheeled robot. http://www.informatics.sussex.ac.uk/users/anils/Teaching/AdSys/low-inertia-robot.ppt*

[15] Song, S., Miller, K.D. & Abbott, L.F. (2000) *Competitive Hebbian learning through spike-timing-dependent synaptic plasticity.* Nature Neuroscience, Vol. 3, pp. 919-926.

[16] Song, S. and Abbott L. *Cortical development and remapping through spike timing-dependent plasticity.* Neuron, Vol. 32(2), pp. 339-350.

[17] Turrigiano, G.G. (1999) *Homeostatic plasticity in neuronal networks: the more things change, the more the stay the same.* Trends in Neurosciences, Vol 2(5), pp. 221-227.

[18] Wang, H.X., Gerkin, R.C., Nauen, D.W. & Bi, G.Q. (2005) *Co-activation and timing-dependent integration of synaptic potentiation and depression.* Natural Neuroscience, Vol. 8, pp.187-193.

[19] Williams, H. (2004) *Homeostatic plasticity in recurrent neural networks.* From Animals to Animats, Proc. of the Eighth Intl. Conf. on Simulation of Adaptive Behavior, .

Appendix A - Study Data

The limits used with the neural controller are:

Term	Description	Min Value	Max Value
τ_i	decay constant	0.4	4.0
b_i	bias	-3.0	3.0
w_{ji}	synaptic weight	-8.0	8.0
$Gain$	sensory and motor gain	0.01	10.0
n_{ij}	rate of plasticity	-0.9	0.9

Table 1: Value limits set in [5].

Appendix B - Differences in experimental approach

Di Paolo used an 8 node CTRNN, in this project, simpler 4 node controllers were used to try and aid understanding of the CTRNN dynamics. This micro-controller could be viewed as making the agents like simple Braintenburg vehicles.

Di Paolo used a Rank selection GA. A microbial GA is used in this model. Rank selection was used initially but the choice of GA seemed to have little impact.

Di Paolo did not mutate if the mutation exceed the bounds [0,1] in the real component of the genotype. In this project, mutations are allowed out of bounds but are truncated within range.

Di Paolo introduces an indeterminacy regarding the exact position of the sensors. The average angles for sensors from the body's midline must be 60º with a uniform variation of ± 5º. For this paper, the sensors will be fixed at 60º from the midline. No mention is made of the sensors' acceptance angle and in this report they're fixed at 80º.

Noise was added to the sensor and motor in Di Paolo's study. Sensor noise (mean = 0, range =0.25) and motor noise (mean = 0.25, range = 0.25). It was found that evolving with noise took a very long time and for reasons of brevity it was removed from the project.

Appendix C - Code

Agent.h

```
/*
 *  Agent.h
 */
#include <vector>

class LightSource;

/**
 * Defines the separation of the sensors
 */
#define kSENSOR_SEPARATION_ANGLE 60.0

/**
 * Defines the acceptance angle of the sensors
 */
#define kSENSOR_ACCEPTANCE_ANGLE 80.0

class Agent
{
        public:
                Agent(Position pos, float radius, float angle);

                // Writes to OpenGL buffer
                void Draw ();

                /**
                 * Return the current position of the agent
                 */
                Position        Pos();

                /**
                 * Return the starting postion of the agent
                 */
                Position        StartPos()                                              {return m_start;}

                void            SetPos (Position& new_pos);

                float           Angle()                                                 {return m_angle;}
                void            SetAngle (float angle);

                /**
                 * Run 'one unit' of time and return agents new position
                 * @return True if agent is ok, false to stop the simulation
                */
                bool            RunOneUnit      (float vr, float vl, float time_step);

                bool            Sense           (std::vector<float>& inputs, LightSource* source);

                /**
                 * For debugging only
                 * Only affects visual state
                 */
                void            SetSensorsActive (bool s1, bool s2);

        private:
                Position m_pos;         //!< current x position (of robot centre)
                Position m_start;       //!< start position of agent
                float m_angle;          //!< current heading angle - degrees    (Alpha)
                float m_sensor;         //!< angle of sensors from mid-line - degrees (Beta)
                float m_radius;         //!< Radius of agent body
```

```
                bool m_s1Active;                //!< True if s1_active;
                bool m_s2Active;                //!< True if s2_active;
                void    DrawWheel       (float offset);
};

float NormaliseAngle (float angle);
```

Agent.cpp

```
/*
 *  Agent.cpp
 */

#include <math.h>
#include "Globals.h"

#if kVISUALISE_AGENT
#include <GLUT/glut.h>
#endif

#if WIN_ENV
#include <float.h>
#endif


#include "Geo.h"
#include "Agent.h"
#include "LightSource.h"
#include "OGLPrimitives.h"

// Angle from horizontal to attach wheels to agent
// 90.0 == side of robot
#define kDRAW_WHEEL_OFFSET 90.0

// Scaling factor used to work out wheel length
// Used for drawing agent only
#define kDRAW_WHEEL_FACTOR 2.5

// Simple conversion factor for converting degrees to rads.
const float     DEG2RAD = kPI / 180.0f;
const float RAD2DEG = 180.0/kPI;

float NormaliseAngle (float angle);


#if WIN_ENV
static bool isnan (float x)
{
        return (_isnan(x) != 0);
}
#endif

//--------------------------------------------------------------------

Agent::Agent(Position pos, float radius, float angle) :
m_pos(pos), m_radius(radius), m_angle(angle), m_sensor(kSENSOR_SEPARATION_ANGLE),
m_start(pos), m_s1Active(false), m_s2Active(false)
{
        SetAngle(angle);
}

//--------------------------------------------------------------------

Position Agent::Pos ()
```

```
{
        return m_pos;
}

//---------------------------------------------------------------------

void Agent::SetSensorsActive (bool s1, bool s2)
{
        m_s1Active = s1;
        m_s2Active = s2;
}

//---------------------------------------------------------------------

void Agent::SetAngle(float angle) {
        m_angle = NormaliseAngle(angle);
}

//---------------------------------------------------------------------
// Calculates the level of light falling on each sensor
bool Agent::Sense (std::vector<float>& inputs, LightSource* source)
{
        inputs[0] = 0.0;
        inputs[1] = 0.0;

        if (source == 0) {
                return false;
        }


        //---- Work out if sensors are active -----------------
        Position light_pos = source->Pos();

        // Calc angle between agent and light
        float x = light_pos.x - m_pos.x;
        float y = light_pos.y - m_pos.y;
        float ang = NormaliseAngle((atan2(y, x) * RAD2DEG) - m_angle);

        float s1_llim = NormaliseAngle(kSENSOR_SEPARATION_ANGLE + kSENSOR_ACCEPTANCE_ANGLE);
        float s1_hlim = NormaliseAngle(kSENSOR_SEPARATION_ANGLE - kSENSOR_ACCEPTANCE_ANGLE);

        float s2_llim = NormaliseAngle(-kSENSOR_SEPARATION_ANGLE + kSENSOR_ACCEPTANCE_ANGLE);
        float s2_hlim = NormaliseAngle(-kSENSOR_SEPARATION_ANGLE - kSENSOR_ACCEPTANCE_ANGLE);

        bool s1 = false;
        bool s2 = false;

        if (ang <= s1_llim) {
                s1 = true;
                if (ang <= s2_llim) {
                        s2 = true;
                }
        }
        else if (ang >= s2_hlim) {
                s2 = true;
                if (ang >= s1_hlim) {
                        s1 = true;
                }
        }

        //---- Turn on display of sensors
        SetSensorsActive(s1, s2);

        float rad, xs, xy, ds, da, ds2;
```

```
        // Work out position of 'top' sensor
        if (s1 == true) {
                rad = (m_angle + m_sensor) * DEG2RAD;
                xs  = m_pos.x + (m_radius * cos (rad));
                xy  = m_pos.y + (m_radius * sin (rad));

                // Calculate distance to sensor from light source
                ds = DistanceBetweenPoints(xs, xy, light_pos.x, light_pos.y);

                // Calculate distance to robot centre and light source
                da = DistanceBetweenPoints(m_pos.x, m_pos.y, light_pos.x, light_pos.y);
                ds2 = ds * ds;

#if 0
                float a = ((m_radius * m_radius) + ds2) / (da * da);

                // light falls on sensor
                if ( a <= 1.0) {
                        inputs[0] = (source->Intensity() / ds2);
                }
#else
                inputs[0] = (source->Intensity() / ds2);
#endif
        }

        // Examine 'bottom' sensor following same approach
        if (s2 == true) {
                rad = (m_angle - m_sensor) * DEG2RAD;
                xs = m_pos.x + m_radius * cos (rad);
                xy = m_pos.y + m_radius * sin (rad);

                ds  = DistanceBetweenPoints(xs, xy, light_pos.x, light_pos.y);
                ds2 = ds * ds;
#if 0
                float a   = ((m_radius * m_radius) + ds2) / (da * da);

                // light falls on sensor
                if ( a <= 1.0) {
                        inputs[1] = (source->Intensity() / ds2);
                }
#else
                inputs[1] = (source->Intensity() / ds2);
#endif
        }
        return true;
}

//-----------------------------------------------------------------------

bool Agent::RunOneUnit (float vr, float vl, float time_step)
{
        float rad = m_angle * DEG2RAD;

        // Work out updates based on new motor values
        float Vc    = (vr + vl) / (float) 2.0;
        float omega = (vr - vl) / (float)(2.0 * m_radius);

        // Calculate change in this time step
        float y_change= time_step * (Vc * sin(rad));
        float x_change= time_step * (Vc * cos(rad));
        float a_change= time_step * omega;

        // Update position
        m_pos.x = m_pos.x + x_change;
        m_pos.y = m_pos.y + y_change;
```

```
        SetAngle(m_angle + a_change);

        if ( isnan(m_pos.x) || isnan(m_pos.y) || isnan(m_angle) ) {
                rad++;
        }
        return true;
}

//---------------------------------------------------------------------
// Writes to OpenGL buffer
// Rewrite using OpenGL rotation transformation
void Agent::Draw ()
{
#if kVISUALISE_AGENT

        // Mark origin
        glColor3f(0.0, 0.0, 1.0);
        DrawCross(kAGENT_RADIUS, m_start.x, m_start.y);

        glColor3f(0.5, 0.5, 0.5);

        float x = m_pos.x;
        float y = m_pos.y;

        // sensor size
        float sensor_size = m_radius / 5.0;

        // Draw body
        DrawLineCircle(m_radius, x, y);

        // Draw 'top' sensor
        glColor3f(0.0, 0.0, 1.0);

        float rad = (m_angle + m_sensor) * DEG2RAD;
        float xs = x + (m_radius * cos (rad));
        float xy = y + (m_radius * sin (rad));
        DrawFilledCircle(sensor_size, xs, xy);

        // Draw active circle
        if (m_s1Active) {
                DrawLineCircle(sensor_size + 2, xs, xy);
        }

        // Draw 'bottom' sensor
        rad = (m_angle - m_sensor) * DEG2RAD;
        xs = x + m_radius * cos (rad);
        xy = y + m_radius * sin (rad);
        DrawFilledCircle(sensor_size, xs, xy);

        // Draw active circle
        if (m_s2Active) {
                DrawLineCircle(sensor_size + 2, xs, xy);
        }

        // find front of agent
        rad = m_angle * DEG2RAD;
        float xf = x + (m_radius * cos (rad));
        float yf = y + (m_radius * sin (rad));
        DrawLine(x, y, xf, yf);

        glColor3f(0.0, 0.0, 0.0);
        // Draw 'wheels'
        DrawWheel (kDRAW_WHEEL_OFFSET);
        DrawWheel (360 - kDRAW_WHEEL_OFFSET);
```

```
#endif
}

//-----------------------------------------------------------------------
// Wheel length is equal to (radius/kWHEEL_FACTOR) * 2
void Agent::DrawWheel (float offset)
{
#if kVISUALISE_AGENT
        glLineWidth(3.0);
        float rad = (m_angle + offset) * DEG2RAD;

        // First calculate anchor point for wheel on robot
        float ax = m_pos.x + m_radius * cos (rad);
        float ay = m_pos.y + m_radius * sin (rad);

        // Highlight attachment point with small circle
        DrawFilledCircle(0.5, ax, ay);

        // Now draw wheel 'horizontal' - ax, ay give anchor point for wheel
        float length = m_radius / (float) kDRAW_WHEEL_FACTOR;
        float x1 = - length;
        float x2 = length;

        // Rotate whee points by body angle
        rad = (m_angle) * DEG2RAD;
        float cos_ang = cos (rad);
        float sin_ang = sin (rad);

        // Translate start point
        float wx1 = (x1 * cos_ang) - (0 * sin_ang) + ax;
        float wy1 = (x1 * sin_ang) + (0 * cos_ang) + ay;

        // Translate end point
        float wx2 = (x2 * cos_ang) - (0 * sin_ang) + ax;
        float wy2 = (x2 * sin_ang) + (0 * cos_ang) + ay;

        glBegin(GL_LINES);
                glVertex2d(wx1, wy1);
                glVertex2d(ax, ay);

                glVertex2d(ax, ay);
                glVertex2d(wx2, wy2);
        glEnd();

        // Restore line width to default
        glLineWidth(1.0);
#endif
}

//-----------------------------------------------------

void Agent::SetPos (Position& new_pos)
{
        m_pos = new_pos;
}

//-----------------------------------------------------

float NormaliseAngle (float angle)
{
        while (angle > 360.0) {
                angle -= 360.0;
        }

        while (angle < 0.0) {
```

```
                angle += 360.0;
        }
        return angle;
}
```

Common.h

```
#pragma once

/*
 * File contains shared bootstrap code between mac and windows
 */

/**
 * Call back function to show progress
 */
typedef bool (*ProgCall) (const char* message, const char* gene, void* data);

/**
 * Shared start call
 */
void Evolve (ProgCall cb, void * data);
```

GLToJPEG.h

```
/*
 *  GLToJPEG.h
 */

/**
 * Export the current OpenGL Buffer as a JPEG file
 * @param       width           The width of the OpenGL buffer
 * @param       height          The height of the OpenGL buffer
 * @param       path            The path to export to
 * @param       quality         The quality of the JPEG image
 * @return      True if succesful, false otherwise.
 */
bool GlExportAsJPEG (unsigned int width, unsigned int height, const char *path, int quality);
```

GLToJPEG.cpp

```
/*
 *  GLToJPEG.cpp
 */
#include <stdlib.h>
#include <stdio.h>

#include "Globals.h"

#if kVISUALISE_AGENT
#include <GLUT/glut.h>
#include "GLToJPEG.h"

//-------------------------------------------------------------

extern "C" {
        #include <jpeglib.h>    /* IJG JPEG LIBRARY by Thomas G. Lane */
}

//-------------------------------------------------------------
/*
        Uses the Independent JPEG Group's free JPEG library to
```

```
        export a GL Buffer as a jpeg image.
 */
bool GlExportAsJPEG (unsigned int width, unsigned int height, const char *path, int quality)
{
        bool ret = false;
        struct jpeg_compress_struct cinfo; // the JPEG OBJECT
        struct jpeg_error_mgr jerr; // error handler struct

        unsigned char *row_pointer[1]; // pointer to JSAMPLE row[s]
        GLubyte *pixels=0, *flip=0;
        FILE *shot;
        int row_stride; // width of row in image buffer

         if((shot=fopen(path, "wb"))!=NULL) { // jpeg file
            // initialization
            cinfo.err = jpeg_std_error(&jerr); // error handler
            jpeg_create_compress(&cinfo); // compression object
            jpeg_stdio_dest(&cinfo, shot); // tie stdio object to JPEG object

            row_stride = width * 3;
            pixels              = (GLubyte *) malloc (sizeof(GLubyte)*width*height*3);
            flip                = (GLubyte *) malloc (sizeof(GLubyte)*width*height*3);

            if ( (pixels != NULL) && (flip != NULL) ) {
               // save the screen shot into the buffer
               //glReadBuffer(GL_FRONT_LEFT);
               glPixelStorei(GL_PACK_ALIGNMENT, 1);
               glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, pixels);

               // give some specifications about the image to save to libjpeg
               cinfo.image_width      = width;
               cinfo.image_height     = height;
               cinfo.input_components = 3; // 3 for R, G, B
               cinfo.in_color_space   = JCS_RGB; // type of image

               jpeg_set_defaults(&cinfo);
               jpeg_set_quality(&cinfo, quality, TRUE);
               jpeg_start_compress(&cinfo, TRUE);

               // OpenGL writes from bottom to top, where as libjpeg goes from
                           // top to bottom. So we need to flip lines.
               for (int y = 0; y < height; y++) {
                  for (int x = 0; x < width; x++) {
                     flip[(y*width+x)*3]   = pixels[((height-1-y)*width+x)*3];
                     flip[(y*width+x)*3+1] = pixels[((height-1-y)*width+x)*3+1];
                     flip[(y*width+x)*3+2] = pixels[((height-1-y)*width+x)*3+2];
                  }
               }

               // write the lines
               while (cinfo.next_scanline < cinfo.image_height) {
                  row_pointer[0] = &flip[cinfo.next_scanline * row_stride];
                  jpeg_write_scanlines(&cinfo, row_pointer, 1);
               }
               ret = true;

               // finish up and free resources
               jpeg_finish_compress(&cinfo);
               jpeg_destroy_compress(&cinfo);
            }
            fclose(shot);
                }

                // Clean-up buffers
                if (pixels != 0) {
```

```
                        free(pixels);
                }
                if (flip != 0) {
                        free(flip);
                }
        return ret;
}

#endif // kVISUALISE_AGENT
```

Genetics.h

```
/*
 * Genetics.h
 */

#pragma once
// TODO: Need to introduce sensor indeterminacy (+/- 5 degrees)

/**
 * Mutation probability at each locus of real section of genotype
 * Use creep mutation which only makes a small change in the value at a loci
 * so mutation rate needs to be high.
 */
#define kPROB_REAL_MUT 0.5

/**
 * Mutation probability at each locus of int section of genotype
 * Use point mutation for ints so small change makes large difference
 * Use a much lower probability than with real values.
 */
#define kPROB_INT_MUT 0.1

/**
 * Size of gaussian vector mutation to add
 */
#define GAUSVECMUT    0.01

/**
 * Recombination probability at each locus
 * May need to be higher apporx = 0.9 which ensures virtual elitism otherwise
 * genotype gets too fragmented
 */
#define kPROB_REC 0.5


/*---------------------------------------------------------------------------
 *
 */

typedef std::vector<int>                        IntGenotype;
typedef std::vector<double>                     RealGenotype;

typedef struct
{
        RealGenotype    rgene;
        IntGenotype             igene;
}
Genotype;

typedef std::vector<Genotype*>          Population;

void RandomisePopulation (Population& pop, int pop_size);
void PrintPopulation     (Population& pop);
void ClearPopulation     (Population& pop);
```

```
void MutateGenotype              (Genotype& winner, Genotype& loser);
void MutateGenotype              (Genotype& winner);


void PrintGenotype              (Genotype& gene);
std::string
        GenotypeToString        (Genotype& gene);

std::string
        GenotypeToDot           (Genotype& gene);
```

Genetics.cpp

Genotype.h

```
/*
 *  Genotype.h
 */



/**
 * A genotype for a fully connected CTRNN
 *
 * Organised into gene blocks each consisting of :
 *              [Time, Bias and Node to Node connection weights]
 * There are n gene blocks (where n is set at Construction time)
 */
class Genotype
{
        typedef enum
        {
                WeightGene,
                TimeGene,
                BiasGene
        }
        GeneType;

public:


        Genotype(int number_of_nodes);

        void MutateGenotype (Genotype& winner);
        void FillFromFile       (std::string fileName);

protected:
        void MutateLocus        (Genotype& winner, int pos, GeneType type);

private:

        // Randomise the genotype
        void Randomise ();

        int m_nodes;                            //!< Number of nodes in genotype
        double_array    m_gene;         //!< Genetic material
};
```

Geo.h

```
/*
 *  Geo.h
 */

#pragma once

typedef struct
{
        float x;                        // position on axis;
        float y;                        // position on yaxis
} Position;

/**
 * Calculate the distance between two 2d points
 */
float DistanceBetweenPoints  (float x1, float y1, float x2, float y2);

void  RotatePointAroundPoint (float& x1, float& y1, float x2, float y2, float angle);

/**
 * Convert a value ranged between [0,1] into a new range
 */
float RescaleUnitValue (float min, float max, float value);
```

Geo.cpp

Globals.h

```
/**
 * Global constants
 */

#pragma once

/**
 * If true, use OpenGL to visualise agent
 */
#if MAC_ENV
#define kVISUALISE_AGENT 1
#else
#define kVISUALISE_AGENT 0
#endif

#define kWindowWidth            1000            //!< Width of the OpenGL window
#define kWindowHeight           750             //!< Height of the OpenGL window


/**
 * Define PI for maths work
 */
#define kPI                             3.14159265358979323846

/**
 * Number of generations to run
 */
#define kNUMBER_OF_GENERATIONS 10

// Value ranges taken from
// Homeostatic adaptation to inversion of the visual field and other sensory motor
// distruptions - Di Paolo unless otherwise stated
```

```
/**
 * Population size
 */
#define kPOPULATION_SIZE 60

/**
 * Di Paolo, E. A. (2000). Behavioral coordination, structural
 * congruence and entrainment in a simulation of acoustically coupled agents.
 * Adaptive Behavior 8:1. 25 - 46 used a body radius of 4.0
 */
#define kAGENT_RADIUS           4.0

/**
 * Integration time step
 */
#define kTIME_STEP                      0.2

/**
 * If true then plasticity rules can be used
 */
#define kALLOW_PLASTIC_CHANGES 0

/**
 * Number of time steps before evaluation starts
 */
#define kNUMBER_OF_PREEVAL_STEPS 50

/**
 * From DiPaolo - min length of time for light to display
 */
#define kMIN_LIGHT_LENGTH 400

/**
 * From DiPaolo - max length of time for light to display
 */
#define kMAX_LIGHT_LENGTH 800

/**
 * Minimum distance 'from agent'
 */
#define kMIN_LIGHT_DISTANCE (10 * kAGENT_RADIUS)

/**
 * Minimum distance 'from agent'
 */
#define kMAX_LIGHT_DISTANCE (25 * kAGENT_RADIUS)

/**
 * Minium angle to place light at
 */
#define kMIN_LIGHT_ANGLE 0.0

/**
 * Maximum angle to plact light at
 */
#define kMAX_LIGHT_ANGLE 360.0

/**
 * From DiPaolo - min value for light intensity
 */
#define kMIN_INTENSITY 500.0

/**
 * From DiPaolo - max value for light intensity
```

```
 */
#define kMAX_INTENSITY 1500.0

/**
 * From Di Paolo - number of lights to present to each agent
 */
#define kNUMBER_LIGHTS_PER_PRESENTATION 5

/**
 * Number of lights to test long term stability over
 */
#define kNUMBER_OF_LONG_TERM_LIGHTS 20

/**
 * From Di Paolo - use a fully connected 8 neuron controller
 */
#define kCTRNN_NUM_NODES 4

/**
 * From Di Paolo - Define max strength of a synaptic connection
 */
#define kMAX_WEIGHT_SIZE 8.0

/**
 * From Di Paolo - Define min strength of a synaptic connection
 */
#define kMIN_WEIGHT_SIZE -8.0

/**
 * From Di Paolo - Define max decay constant of a node
 */
#define kMAX_TIME_SIZE 4.0

/**
 * From Di Paolo - Define min decay constant of a node
 */
#define kMIN_TIME_SIZE 0.4

/**
 * From Di Paolo - Define max bias constant of a node
 */
#define kMAX_BIAS_SIZE 3.0

/**
 * From Di Paolo - Define min bias constant of a node
 */
#define kMIN_BIAS_SIZE -3.0

// Scalars on motor output
#define kMAX_MOTOR_GAIN 10.0
#define kMIN_MOTOR_GAIN 0.01

// Scalars on sensory input
#define kMAX_INPUT_GAIN 10.0
#define kMIN_INPUT_GAIN 0.01

// Rate of change for plasticity
#define kMAX_RATE_OF_CHANGE 0.9
#define kMIN_RATE_OF_CHANGE -0.9

// Length of real genotype
#define kREAL_GENOTYPE_LENGTH ( 2 + kCTRNN_NUM_NODES + kCTRNN_NUM_NODES + (2 * (kCTRNN_NUM_NODES * kCTRNN_NUM_NODES)) )

// Length of int genotype
#define kINT_GENOTYPE_LENGTH  (kCTRNN_NUM_NODES * kCTRNN_NUM_NODES)
```

```
/**
 * If true, noise is added to sensor and motor inputs
 */
#define kEVOLVE_WITH_NOISE 0
```

LightSource.h

```
/*
 *  LightSource.h
 */

#pragma once

// If true then light sources are left for kMAX_LIGHT_LENGTH
#define kLONG_LIGHT_SOURCE 0

// Defines a light source
class LightSource
{
public:
        /**
         * Will place lightsource 'near' position
         */
        LightSource(const Position& pos);

        /**
         * Will place lightsource 'near' position at given angle
         */
        LightSource(const Position& pos, float angle);

        virtual ~LightSource()                          {}

        /**
         * Will place lightsource at random position
         */
        LightSource();

        /**
         * Issue OpenGL commands to draw self
         */
        virtual void    Draw    ();

        Position        Pos() const;
        float           Intensity()     const   {return m_intensity;}

        /**
         * Returns the length of time to display the light
         * To get integration steps / kTIME_STEP
         * @returns The simulation time for the light
         */
        int                     Time() const            {return m_counter;}

protected:
        int             m_counter;

private:
        float   m_x;            // position of centre
        float   m_y;            // position
        float   m_intensity;

        void SharedInit ();
};
```

```
class LongLightSource : public LightSource
{
public:
        LongLightSource(const Position& pos);
        ~LongLightSource() {}
};
```

LightSource.cpp

```
/*
 *  LightSource.cpp
 */

#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include "Globals.h"
#include "Geo.h"
#include "LightSource.h"
#include "Random.h"
#include "OGLPrimitives.h"

/**
 * Used to display lightsource intensity
 */
#define kLIGHT_SOURCE_SCALING_THRESHOLD 12.0

//-----------------------------------------------------------

LightSource::LightSource(const Position& pos)
{

        m_x = pos.x + (float) RandomLib::RangedDouble(kMIN_LIGHT_DISTANCE, kMAX_LIGHT_DISTANCE);
        m_y = pos.y + (float) RandomLib::RangedDouble(kMIN_LIGHT_DISTANCE, kMAX_LIGHT_DISTANCE);

        float angle = (float) RandomLib::RangedDouble(kMIN_LIGHT_ANGLE, kMAX_LIGHT_ANGLE);

        RotatePointAroundPoint(m_x, m_y, pos.x, pos.y, angle);

        SharedInit();
}

//-----------------------------------------------------------
// special light source fixed in front of agent
LightSource::LightSource(const Position& pos, float angle)
{

        m_x = pos.x + (float) RandomLib::RangedDouble(kMIN_LIGHT_DISTANCE, kMAX_LIGHT_DISTANCE);
        m_y = pos.y;

        SharedInit();
}

//-----------------------------------------------------------

LightSource::LightSource() : m_counter(0)
{
        float range = kWindowWidth / 2.0;
        m_x = (float) RandomLib::RangedDouble(-range, range);

        range = kWindowHeight / 2.0;
        m_y = (float) RandomLib::RangedDouble(-range, range);

        SharedInit();
}
```

```
//-------------------------------------------------------

Position LightSource::Pos () const
{
        Position pos = {m_x, m_y};
        return pos;
}

//-------------------------------------------------------

void LightSource::SharedInit ()
{
        // Calculate the number of steps the light source should stick around for
#if kLONG_LIGHT_SOURCE
        T = kMAX_LIGHT_LENGTH;
#else
        float T = kMIN_LIGHT_LENGTH;
#endif

        // Set up length of time for light_source
        // 0.75 and 1.25 are from the Di Paolo paper

        // TOOD: Need to pay attention to whats going on when we
        // use a different time step from DiPaolo
        float tmin = 0.75 * T;
        float tmax = 1.25 * T;
        m_counter = (int) RandomLib::RangedDouble(tmin, tmax);

        assert(m_counter > 0);

        // As per paper
        m_intensity = (float) RandomLib::RangedDouble(kMIN_INTENSITY, kMAX_INTENSITY);
}

//-------------------------------------------------------

void LightSource::Draw()
{
#if kVISUALISE_AGENT

        // Show strength of light
        DrawLineCircle(m_intensity / kLIGHT_SOURCE_SCALING_THRESHOLD, m_x, m_y);

        // Use kAGENT_RADIUS for radius
        DrawLightSource(kAGENT_RADIUS, m_x, m_y);
#endif
}


//-------------------------------------------------------
// LongLightSource methods
//-------------------------------------------------------

LongLightSource::LongLightSource(const Position& pos) : LightSource(pos)
{
        m_counter = 4000;                       // v. stable light source
}
```

main.cpp

```
/*
 * Main.cpp
 */
```

```
#include <string>
#include <iostream>
#include <vector>
#include <GLUT/glut.h>

#include "Globals.h"
#include "Geo.h"
#include "Agent.h"
#include "OGLPrimitives.h"
#include "Random.h"
#include "Simulation.h"
#include "LightSource.h"
#include "GLToJPEG.h"

#include "Math.h"

//--------------------------------------------------------------------------
// Global simulation object
Simulation* gSim  = 0;

// Global agent object
Agent*      gAgent= 0;

// Global light source
LightSource*    gSource = 0;

float           gAngle = 0.0;

typedef enum
{
        kNothing,
        kPause,
        kEvol,
        kRun,
        kAgentDraw
}
WorldState;

// Start in draw state
WorldState gState = kAgentDraw;
WorldState gPriorState = gState;

float gZoom = 2.5;

//--------------------------------------------------------------------------
/*
 * Code for the Right Click Menu
 */
void Glut_Menu (int id)
{
        // Examine the request
        switch (id) {

                case 1:
                {
                        gState = kEvol;
                        gSim->Init();
                }
                break;

                case 2:
                {
                        gState = kRun;
                        std::string fileName = kDEFAULT_FILE_PATH;
                        fileName += kDEFAULT_CHAMPION_AGENT_FILE;
```

```
        gSim->StepInit(fileName, kNUMBER_LIGHTS_PER_PRESENTATION, false, false);
}
break;

case 3:
{
        gState = kRun;
        std::string fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_END_AGENT_FILE;
        gSim->StepInit(fileName, kNUMBER_LIGHTS_PER_PRESENTATION, false, false);
}
break;

case 4:
{
        // Check long term stability
        gState = kRun;
        std::string fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_CHAMPION_AGENT_FILE;
        gSim->StepInit(fileName, kNUMBER_OF_LONG_TERM_LIGHTS, false, false);
}
break;

case 5:
{
        gState = kRun;
        std::string fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_END_AGENT_FILE;
        gSim->StepInit(fileName, kNUMBER_OF_LONG_TERM_LIGHTS, false, false);
}
break;

case 6:
{
        // Check stability in presence of inversion
        gState = kRun;
        std::string fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_CHAMPION_AGENT_FILE;
        gSim->StepInit(fileName, kNUMBER_OF_LONG_TERM_LIGHTS, true, false);
}
break;

case 7:
{
        gState = kRun;
        std::string fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_END_AGENT_FILE;
        gSim->StepInit(fileName, kNUMBER_OF_LONG_TERM_LIGHTS, true, false);
}
break;

case 8:
{
        gState = kRun;
        std::string fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_CHAMPION_AGENT_FILE;
        gSim->StepInit(fileName, kNUMBER_LIGHTS_PER_PRESENTATION, true, true);
}
break;

case 9:
{
        gState = kRun;
        std::string fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_END_AGENT_FILE;
```

```
                        gSim->StepInit(fileName, kNUMBER_LIGHTS_PER_PRESENTATION, true, true);
                }
                break;

                case 10:
                {
                        gState = kAgentDraw;
                }
                break;

                case 11:
                {
                        // Save buffer
                        // @todo: Provide ui to change path
                        std::string file = kDEFAULT_FILE_PATH;
                        file += "Image.jpg";
                        GlExportAsJPEG(kWindowWidth, kWindowHeight, file.c_str(), 80);
                }
                break;

                case 0:
                {
                        // User has asked to quit
                        exit(0);
                }
                break;
        }
}

//--------------------------------------------------------------------

void HandleKeyPress (unsigned char key, int x, int y)
{
        switch (key)
        {
                case 'p':
                        if (gState == kPause) {
                                gState = gPriorState;
                        }
                        else {
                                gPriorState = gState;
                                gState = kPause;
                        }
                        break;

                case'r' :
                        if (gAgent && (gState == kAgentDraw)) {
                                float angle = gAgent->Angle();
                                angle += 5.0;
                                gAgent->SetAngle(angle);
                        }
                        break;

                case'f' :
                        if (gAgent && (gState == kAgentDraw)) {
                                Position pos = gAgent->Pos();
                                if (rand() % 1) {
                                        pos.x -= rand() % 2; pos.y--;
                                }
                                else {
                                        pos.x += rand() % 2; pos.y++;
                                }
                                gAgent->SetPos(pos);
                        }
                        break;
```

```
                case '+' :
                        {
                                gZoom += 0.5;
                                float xrange = kWindowWidth / gZoom;
                                float yrange = kWindowWidth / gZoom;
                                glOrtho(-xrange, xrange, -yrange, yrange, -1.0, 1.0);
                        }
                        break;

                case '-' :
                        {
                                gZoom -= 0.5;
                                float xrange = kWindowWidth / gZoom;
                                float yrange = kWindowWidth / gZoom;
                                glOrtho(-xrange, xrange, -yrange, yrange, -1.0, 1.0);
                        }
                        break;

                default:
                        break;
        }
        // just dump key out to log
        std::cout << key;
}

//---------------------------------------------------------------------
// Start the OpenGL Utility Toolkit (GLUT) and create a window to draw in
int InitGLUT (int argc, char** argv, std::string message)
{
        // Start glut - will extract any glut specific parameters from
        // the command line
        glutInit(&argc, argv);

        // Set the display mode to: double buffered window, using RGB and
        // depth buffer
        glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

        // Set the window size we're using to draw
        glutInitWindowSize (kWindowWidth, kWindowHeight);

        // Set the default display position of the window
        glutInitWindowPosition (kWindowWidth / 2, kWindowWidth / 2);

        // Create the display window using the passed title.
        int id = glutCreateWindow (message.c_str());

        // Create menu
        glutCreateMenu(Glut_Menu);
        glutAddMenuEntry("Evolve", 1);
        glutAddMenuEntry("Study Elite Champion", 2);
        glutAddMenuEntry("Study End Champion", 3);
        glutAddMenuEntry("Stablibity of Elite Champion", 4);
        glutAddMenuEntry("Stabibility of End Champion", 5);
        glutAddMenuEntry("Invert Elite Champion", 6);
        glutAddMenuEntry("Invert End Champion", 7);
        glutAddMenuEntry("Mini-Invert Elite Champion", 8);
        glutAddMenuEntry("Mini-Invert End Champion", 9);
        glutAddMenuEntry("Agent Draw Test", 10);
        glutAddMenuEntry("Save Screen as JPEG", 11);
        glutAddMenuEntry("Quit", 0);
        glutAttachMenu(GLUT_RIGHT_BUTTON);

        // Register callback
        glutKeyboardFunc(HandleKeyPress);
```

```
        return id;
}

//------------------------------------------------------------------------
// Draws the basic agent
void DisplayWorld (void)
{
        // Set up OpenGL area
        glClearColor(1.0, 1.0, 1.0, 0.0);
        glClear(GL_COLOR_BUFFER_BIT);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        float xrange = kWindowWidth / gZoom;
        float yrange = kWindowWidth / gZoom;
        glOrtho(-xrange, xrange, -yrange, yrange, -1.0, 1.0);

        gSim->Draw();

        if (gState == kPause) {
                // Display if pause on or off
                DrawText("Paused", 0.0, 0.0, 18);
        }

        glutSwapBuffers();
}

double CalcAngle(double p1_x, double p1_y, double p2_x, double p2_y)
{
        double a_x = p2_x - p1_x;
        double a_y = p2_y - p1_y;

        double b_x = 1.0;
        double b_y = 0.0;

        return acos((a_x*b_x+a_y*b_y)/sqrt(a_x*a_x+a_y*a_y));
}

//------------------------------------------------------------------------

const float     DEG2RAD = kPI / 180.0f;
const float RAD2DEG = 180.0/kPI;

void CalcSensors (void)
{
        Position lpos = gSource->Pos();
        Position apos = gAgent->Pos();
        float angle = gAgent->Angle();

        float x = lpos.x - apos.x;
        float y = lpos.y - apos.y;
        float ang = NormaliseAngle((atan2(y, x) * RAD2DEG) - angle);

        float s1_llim = NormaliseAngle(kSENSOR_SEPARATION_ANGLE + kSENSOR_ACCEPTANCE_ANGLE);
        float s1_hlim = NormaliseAngle(kSENSOR_SEPARATION_ANGLE - kSENSOR_ACCEPTANCE_ANGLE);

        float s2_llim = NormaliseAngle(-kSENSOR_SEPARATION_ANGLE + kSENSOR_ACCEPTANCE_ANGLE);
        float s2_hlim = NormaliseAngle(-kSENSOR_SEPARATION_ANGLE - kSENSOR_ACCEPTANCE_ANGLE);

        bool s1 = false;
        bool s2 = false;

        if (ang <= s1_llim) {
                s1 = true;
```

```
                if (ang <= s2_llim) {
                        s2 = true;
                }
        }
        else if (ang >= s2_hlim) {
                s2 = true;
                if (ang >= s1_hlim) {
                        s1 = true;
                }
        }

        gAgent->SetSensorsActive(s1, s2);
}

//------------------------------------------------------------------------

void Idle (void)
{
        switch (gState)
        {
                case kRun:
                        {
                                // Just try and redisplay. Clear the depth buffer
                                // as well this is required, otherwise the depth buffer
                                // gets filled and nothing gets rendered.
                                glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

                                if (gSim->StepIntegrate()) {
                                        // Display world
                                        gSim->Draw();
                                }
                                else {
                                        DrawText("Completed", 0.0, 0.0, 18);
                                }
                        }
                        break;

                case kPause:
                        {
                                // Don't clear buffer as we want to overlay text over whats displayed
                                // Display if pause on or off
                                DrawText("Paused", 0.0, 0.0, 18);
                        }
                        break;

                case kEvol:
                        {
                                // Go and make a cup of tea
                                gSim->Init();
                                gSim->RunMicrobial();

                                // Now load the best agent - End champion
                                std::string fileName = kDEFAULT_FILE_PATH;
                                fileName += kDEFAULT_END_AGENT_FILE;

                                // Now display end agent
                                gSim->StepInit(fileName, kNUMBER_LIGHTS_PER_PRESENTATION, false, false);
                                gState = kRun;
                        }
                        break;

                case kAgentDraw:
                        {
                                glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                                gAgent->Draw();
```

```
                                gSource->Draw();

#if DEBUG
                                CalcSensors();
#endif
                        }
                        break;
        }
        glutSwapBuffers();
}

//--------------------------------------------------------------------

int main (int argc, char** argv)
{
        // ID should be positive
        int id = InitGLUT(argc, argv, "Phototaxis");

        // Initialise random lib
        RandomLib::Init();

        // Initialise Simulation
        gSim = new Simulation(kPOPULATION_SIZE);

        // Initalise agent and light source for debug testing
        Position start_pos = {0.0, 0.0};
        gAngle = static_cast<float>(RandomLib::Double() * 360.0);
        gAgent = new Agent(start_pos, kAGENT_RADIUS, gAngle);
        gSource = new LightSource(start_pos);

#if 0
        gSim->Init();
        gSim->RunMicrobial();

        // Now load the best agent - End champion
        std::string fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_END_AGENT_FILE;
        gSim->StepInit(fileName);
#endif

        // Show Agent's path
        if (id >= 1) {
                glutDisplayFunc(DisplayWorld);
                glutIdleFunc(Idle);
                glutMainLoop();
        }

        return 0;
}
```

OGLPrimitives.h

```
/*
 *  OGLPrimitives.h
 */
#pragma once

// Various OpenGL functions for drawing basic primitive shapes

void DrawLineCircle             (double radius, double x, double y);
void DrawFilledCircle   (double radius, double x, double y);

void DrawLineRect               (float x1, float y1, float x2, float y2);
void DrawFilledRect             (float x1, float y1, float x2, float y2);
```

```
void DrawLightSource    (float radius, float x, float y);

void DrawLine                   (float x1, float y1, float x2, float y2);

void DrawText                   (const char *string, float x, float y, int size);
void DrawCross                  (float size, float x, float y);
```

OGLPrimitives.cpp

```
/*
 *  OGLPrimitives.cpp
 */

#include <string>

#include "globals.h"

#if kVISUALISE_AGENT

#include <Glut/Glut.h>
#include "Math.h"
#include "OGLPrimitives.h"

#define DEGTORAD( x ) ( x / 57.29577957795135)
//---------------------------------------------------------------------------
// Modified from
// http://www.gmonline.demon.co.uk/cscene/CS6/CS6-06.html
void DrawLineCircle (double radius, double x, double y)
{
        glBegin(GL_LINE_LOOP);

        for (int i=0; i < 360; i++) {
                float degInRad = DEGTORAD(i);
                glVertex2f((cos(degInRad)*radius) + x, (sin(degInRad)*radius) + y);
        }
        glEnd();
}

//---------------------------------------------------------------------------

void DrawLine (float x1, float y1, float x2, float y2)
{
        glBegin(GL_LINES);
                glVertex2d(x1, y1);
                glVertex2d(x2, y2);
        glEnd();
}

//---------------------------------------------------------------------------
// Draw a circle as a series of filled triangles
void DrawFilledCircle (double radius, double x, double y)
{
        float y1 = y;
        float x1 = x;
        float angle;
        float x2, y2;

        glBegin(GL_TRIANGLES);
        for(int i = 0; i <= 360; i++)
        {
                angle = DEGTORAD(i);
                x2 = x + (radius * (float)sin((double)angle));
                y2 = y + (radius * (float)cos((double)angle));
                glVertex2d(x, y);
                glVertex2d(x2, y2);
```

```
                glVertex2d(x1, y1);
                y1 = y2;
                x1 = x2;
        }
        glEnd();
}

//-----------------------------------------------------------------------

void DrawLineRect (float x1, float y1, float x2, float y2)
{
        glBegin(GL_LINES);
                glVertex2d(x1, y1);
                glVertex2d(x1, y2);

                glVertex2d(x2, y1);
                glVertex2d(x2, y2);


                glVertex2d(x1, y1);
                glVertex2d(x2, y1);


                glVertex2d(x1, y2);
                glVertex2d(x2, y2);
        glEnd();
}

//-----------------------------------------------------------------------

void DrawFilledRect (float x1, float y1, float x2, float y2)
{
        glRectf(x1, y1, x2, y2);
}

//-----------------------------------------------------------------

void DrawText (const char *string, float x, float y, int size)
{
        std::string msg = string;

        // Move raster pos
        glRasterPos2f(x, y);

        // Switch to black ink
        glColor3f(0.0f, 0.0f, 0.0f);

        void* font = GLUT_BITMAP_HELVETICA_10;
        if (size > 10) {
                if (size <=12) {
                        font = GLUT_BITMAP_HELVETICA_12;
                }
                else {
                        font = GLUT_BITMAP_HELVETICA_18;
                }
        }

        std::string::iterator pos = msg.begin();
        for ( ; pos != msg.end(); ++pos) {
                // Draw Text
                glutBitmapCharacter(font, (char)(*pos));
        }
}

//-----------------------------------------------------------------------
```

```
void DrawLightSource (float radius, float x, float y)
{
        glColor3f(1.0, 0.0, 0.0);

        glBegin(GL_LINES);

        float x1, y1;
        float rads;

        // All lines lie in the xy plane.
        int step_angle = 10;    // amount to move on
        if (radius < 10)
                step_angle = 20;

        for(int angle = 0; angle <= 180; angle += step_angle)
        {
                // Top half of the circle
                rads = DEGTORAD(angle);
                x1 = x + (radius * sin(rads));
                y1 = y + (radius * cos(rads));
                glVertex2d(x1, y1);    // First endpoint of line

                // Bottom half of the circle
                rads = angle + 180;
                rads = DEGTORAD(rads);
                x1 = x + (radius * sin(rads));
                y1 = y + (radius * cos(rads));
                glVertex2d(x1, y1);    // Second endpoint of line
        }

        // Done drawing points
        glEnd();
}

//-----------------------------------------------------------------------
// Picks up current colour for drawing
void DrawCross (float size, float x, float y)
{
        glBegin(GL_LINES);

                float add = size / 2.0;

                // cross 1
                glVertex2d(x - add, y + add);
                glVertex2d(x + add, y - add);

                // cross 2
                glVertex2d(x - add, y - add);
                glVertex2d(x + add, y + add);
        // Done drawing points
        glEnd();
}

#endif // kVISUALISE_AGENT
```

Random.h

```
/*
 *  Random.h
 */

class RandomLib
{
public:
```

```
        /**
         * Must call init before doing anything else
         */
        static  void    Init    ();

        static int              RangedInt               (int min, int max);
        static double   Double                  ();
        static double   RangedDouble     (double min, double max);
        static double   RangedGausDouble (double min, double max, double sigma, double centre);

private:
        static  bool    m_init;
};
```

Random.cpp

```
/*
 *  Random.cpp
 */

#include <assert.h>
#include <stdlib.h>
#include <Math.h>
#include "Random.h"
#include <time.h>

bool RandomLib::m_init = false;

// Windows doesn't have any double random number generators
#if WIN_ENV
/*
 * Simple double gnerator - will give doubles in range 0.0 - 1.0
 */
double SimpleUniform()
{
        return (double)rand() / RAND_MAX;
}


/*-------------------------------------------------------------------------
 * Multplicative Linear Congruential Generator
 */
double Uniform()
{
        // Grab random seeds
        int s1 = rand();
        int s2 = rand();

        int z,k;
        k = s1 / 53668;
        s1 = 40014 * (s1 - k * 53668) - k * 12211;

        if (s1 < 0)
                s1 = s1 + 2147483563;
        k = s2 / 52774;
        s2 = 40692 * (s2 - k * 52774) - k * 3791;

        if (s2 < 0)
                s2 = s2 + 2147483399;
        z = s1 - s2;

        if (z < 1)
                z = z + 2147483562;
        return z * 4.65661305956E-10;
}
```

```
/*-------------------------------------------------------------------------
 * .Net/Visual Studio doesn't have drand48 available so provide own
 */
double drand48 ()
{
        return Uniform();
}

#endif

/*-------------------------------------------------------------------------
 *
 */
void RandomLib::Init ()
{
        if (!m_init) {
                srand((unsigned int)time(NULL));
                m_init = true;
        }
}

/*-------------------------------------------------------------------------
 *
 */
int RandomLib::RangedInt(int min, int max)
{
        return ( rand() % (max+1) );
}

/*-------------------------------------------------------------------------
 *
 */
double RandomLib::Double ()
{
        return drand48();
}

/*-------------------------------------------------------------------------
 * Generates a random double in a given range based on a gaussian distribution
 */
double RandomLib::RangedGausDouble (double min, double max, double sigma, double centre)
{
        double random   = (min + (max-min) * (double) rand()/RAND_MAX); //create random domain between [min,max]

        double tmp              = (random-centre) / sigma;
        double gauss    = exp(-tmp*tmp/2); //gaussian formula

        return gauss;
}


/*-------------------------------------------------------------------------
 * Generates a random gene in a given range
 */
double RandomLib::RangedDouble (double min_num, double max_num)
{
        double val = drand48() * (max_num-min_num) + min_num;

        assert ( val >= min_num);
        assert ( val <= max_num);

        return val;
}
```

Simulation.h

```
/*
 *  Simulation.h
 */
#pragma once

#include <vector>
#include "Genetics.h"
#include "SimulationRecord.h"

// Forward ref
class Agent;
class LightSource;

#if WIN_ENV
#define kDEFAULT_FILE_PATH "H:\\r\\rd\\rdp24\\WindowsProfile\\My Documents\\";
#else
#define kDEFAULT_FILE_PATH "/Users/pip/Desktop/Results/"
#endif

#define kDEFAULT_END_AGENT_FILE                 "FitAgent.txt"
#define kDEFAULT_FITNESS_FILE                   "Fitness.txt"
#define kDEFAULT_CHAMPION_AGENT_FILE    "EliteAgent.txt"


/**
 * Simulation object
 */
class Simulation
{
        typedef std::vector<LightSource*>       LightSources;
public:
        Simulation(int population);
        ~Simulation();

        /**
         * Initialises GA for batch processing
         */
        bool Init ();

        /**
         * Find best solution via Microbial GA
         */
        bool RunMicrobial ();

        /**
         * Find best solution via Rank based GA
         */
        bool  RunRank ();

        /**
         * Load and study a single saved genotype
         * @param fileName                      Name of the genotype to load
         * @param numberoftrials        Number of light sources to test
         * @param invert                        Invert sensors during trial
         * @param agent                         Agent is stationary
         */
        bool StepInit (std::string filename, int numberoftrails, bool invert, bool stationary);

        /**
         * Step integrate loaded genotype
         */
        bool StepIntegrate ();
```

```
        /**
         * Called by OpenGL to draw the state of the world
         */
        void Draw ();

#if WIN_ENV
        /**
         * Add progress callback
         */
        void AddProgCB(ProgCall prog, void * data);
#endif

protected:
        double  EvaluateGenotype        (Genotype* geno);
        double  Light                           ();

        // Saves population to given file
        void    SavePopulation          (std::string fileName);

        // Overwrites current population with one from file
        void    ReadPopulation          (std::string fileName);

        /**
         * Saves a particular population member to a file in a text format
         * @param fileName      The file path to save to
         * @param id            The index into the population to save
         * @param asDot         If true save agent in 'dot' format to generate graph
         */
        void    SavePopulationMember (std::string fileName, int id);

        /**
         * Saves a particular population member to a file in '.dot' format
         * @param fileName      The file path to save to
         * @param id            The index into the population to save
         */
        void    ExportPopulationMemberAsDot(std::string fileName, int id);

        // Load a new population member
        Genotype*
                        LoadPopulationMember(std::string fileName);

        // Loads an individual - overwrites current agent
        void    LoadFitess                      (std::string fileName);

        bool    UpdateWeights           (SimulationRecord& rec, DoubleVector& firing_rates);

        /**
         * Integrate current agent for n-steps
         * @returns the number of steps agent spent near the light
         */
        bool Integrate (SimulationRecord& sim, int steps, bool log = false);

        /**
         * Prerun the CTRNN to allow it to settle from random starting
         * position
         */
        bool PreIntegrate       (SimulationRecord& sim);

        /**
         * Generate a sequence of lights for testing
         */
        void GenerateLights (Position pos);

        /**
         * Generate Fitness measure for an agent
```

```
         * Returns overvall value
         * d - [Filled by method] - Distance fitness component
         * h - [Filled by method] - Homeostatic fitness component
         * n - [Filled by method] - Nearness to source fitness component
         */
        float CalcFitness (SimulationRecord& rec, float& d, float& n, float& h);

        /**
         * Clear light entrys
         */
        void ClearLights        ();

        /**
         * Starts logging info about the simulation
         */
        void StartLogs ();

        /**
         * Logs information from the simulation record to the global log
         */
        void ActivityLog (SimulationRecord& rec, DoubleVector& firing_rates);

        /**
         * Logs information from the simulation record to the global log
         */
        void FitnessLog (SimulationRecord& rec, float overall, float distance, float near, float homeo);

        /**
         * Logs information from the simulation record to the global log
         */
        void DistanceLog (SimulationRecord& rec, float distance);

        /**
         * Log information about weight change
         */
        void WeightsLog (SimulationRecord& rec);

        /**
         * Closes open logs
         */
        void EndLogs ();

        /**
         * Save the highest scoring member of the population
         */
        void SaveElite (int elite);

private:
        Agent*                          m_agent;                //!< Agent running in the world
        LightSources            m_sources;
        int                                     m_pop;                  //!< Size of population
        Population*                     m_genes;
        SimulationRecord*
                                                m_rec;
        std::string                     m_buffer;               //!< status text buffer
#if WIN_ENV
        ProgCall                        m_prog;                 //!< Progress callback
        void*                           m_data;                 //!< Data to associate with callback
#endif
};
```

Simulation.cpp

```
/*
 *  Simulation.cpp
```

```
 */

#include <iostream>
#include <iterator>
#include <fstream>
#include <sstream>
#include <map>
#include <math.h>
#include <algorithm>
#include <stdexcept>
#include <string>
#include <vector>


/**
 * Contribution to overall fitness for being near
 * the light at end of the trail
 */
#define kALPHA_MODIFIER 0.2

/* Contribution to overall fitness for being near
 * the light at during the trail
 */
#define kBETA_MODIFIER  0.64

/** Contribution to overall fitness for having
 * stable neurons
 */
#define kGAMMA_MODIFIER 0.16

using namespace std;

#if kVISUALISE_AGENT
#include <Glut/Glut.h>
#endif

#if WIN_ENV
#include <float.h>
#include "Common.h"
#endif

#include "Globals.h"
#include "Genetics.h"
#include "Geo.h"
#include "Agent.h"
#include "LightSource.h"
#include "SimulationRecord.h"
#include "Simulation.h"
#include "Random.h"

#if kVISUALISE_AGENT
#include "OGLPrimitives.h"
#endif

//--- Local Definitions ---------------------------
// Look up table for sigmoid function
// Based on Eduardo's example CTRNN code
double SIGMOID[10000];                          //look-up table for sigmoid function

//---- Local Definitions --------------------------

float Sigmoid                     (float x);
void  PrintVector                 (FloatVector& fit, const char* title);
void  InitSimulationRecord (SimulationRecord& rec, Genotype& gene);
float FloatVectorMean      (FloatVector& fit, int n);
```

```
float FloatVectorStdDev    (FloatVector& fit, float mean, int n);
void  DumpFitness          (float* vect1, float* vect2, float* vect3, int length, std::string fileName, const char* title);

#if WIN_ENV
bool isnan (float x)
{
        return (_isnan(x) != 0);
}
#endif

// Log Files
#define kDEFAULT_ACTIVITY_LOG "ActivityLog.txt"
std::ofstream aLog;                     // Activation, sensor and value log

#define kDEFAULT_DISTANCE_LOG "DistanceLog.txt"
std::ofstream dLog;                     // Distance to source log

#define kDEFAULT_FITNESS_LOG "FitnessLog.txt"
std::ofstream fLog;                     // Fitness component log

#define kDEFAULT_WEIGHT_LOG "WeightLog.txt"
std::ofstream wLog;                     // Weight component log

//-----------------------------------------------

Simulation::Simulation(int pop)  :
m_agent(0), m_pop(pop), m_genes(0), m_rec(0)
{
#if WIN_ENV
        m_prog = 0;
        m_data = 0;
#endif
}

//-----------------------------------------------

Simulation::~Simulation()
{
        if (m_genes) {
                ClearPopulation(*m_genes);
                delete m_genes;
        }
        delete m_rec;
        delete m_agent;
        ClearLights();
}

//-----------------------------------------------

bool Simulation::Init ()
{
        // Generate sigmoid look-up table to save time
        for (int i = 0; i < 10000; i++) {
                SIGMOID[i] = 1 / (1 + exp(-((double)(i-5000)/500)));
        }

        // Randomise population
        if (m_genes) {
                ClearPopulation(*m_genes);
                delete m_genes;
        }
        m_genes = new Population;
        RandomisePopulation(*m_genes, m_pop);

        PrintPopulation(*m_genes);
```

```
        // Set current agent to NULL
        delete m_agent;
        m_agent = 0;

        // delete any attached simulation record
        delete m_rec;
        m_rec = 0;
        return true;
}

//-----------------------------------------------

double Simulation::EvaluateGenotype(Genotype* gen)
{
        if (gen == 0) {
                return 0.0;
        }

        float start_distance, near, run_fitness, end_distance, homeostatic;
        float pos_fitness;

        float overall = 0.0;
        vector<float> fitness (kNUMBER_LIGHTS_PER_PRESENTATION);

        SimulationRecord rec;
        InitSimulationRecord(rec, *gen);

        // Assign random start point
        Position start_pos = {0.0, 0.0};

        // Now we're ready to evaluate genotype
        int light_num = 0;

        for (light_num = 0; light_num < kNUMBER_LIGHTS_PER_PRESENTATION; light_num++) {
                LightSource* source = new LightSource(start_pos);
                m_agent             = new Agent(start_pos, kAGENT_RADIUS,  static_cast<float>(RandomLib::Double() * 360));

                // Prerun agent to allow it settle
                if (!PreIntegrate(rec)) {
                        rec.failed = true;
                }

                Position lpos   = source->Pos();
                start_distance = DistanceBetweenPoints(lpos.x, lpos.y, 0.0, 0.0);
                near           = 0.0;
                pos_fitness    = 0.0;
                run_fitness    = 0.0;
                homeostatic        = 0.0;

                // Initialise counters
                rec.near        = 0;
                rec.homeostatic = 0;

                // Integrate network
                int  steps = source->Time() / kTIME_STEP;

                rec.source = source;
                Integrate(rec, steps);

                if (rec.failed) {
                        fitness[light_num] = 0.0;       // on this run
                }
                else {
                        // distance between agent and light
```

```
                        Position end_pos = m_agent->Pos();
                        end_distance = DistanceBetweenPoints(lpos.x, lpos.y, end_pos.x, end_pos.y);

                        // has agent moved nearer the light?
                        pos_fitness = (float) (1.0 - (end_distance / start_distance));

                        if (pos_fitness < 0.0) {
                                pos_fitness = 0.0;
                        }

                        // Proportion of time spent near the light
                        near = rec.near / (float) steps;

                        // Time average percentage of neurons behaving homeostatically
                        homeostatic = (rec.homeostatic) / ((float) steps * kCTRNN_NUM_NODES);

                        // Average of end point and time spent at the light source
                        run_fitness = (kALPHA_MODIFIER * pos_fitness) + (kBETA_MODIFIER * near) + (kGAMMA_MODIFIER * homeostatic
                }

                // Accumulate
                fitness[light_num] = run_fitness;

                // delete agent
                delete m_agent;
                delete source;
                m_agent = 0;
        }

        // return average fitness
        overall = FloatVectorMean(fitness, light_num);

        // Minus 20% of the standard deviation
        overall = overall - (float) (FloatVectorStdDev(fitness, overall, light_num) * 0.2);
        return overall;
}

//---------------------------------------------------------------------

double Plasticity (double max, double min, double y, double b)
{
        double activity = y + b;
        double p = 0.0;
        if (activity < -max)
                p = -1;
        else if (activity > max) {
                p = 1;
        }

        // otherwise return linear increase in plasticity
        else if (activity < -min) {
                p = (0.5 * activity) + 1;
        }
        else if (activity > min) {
                p = (0.5 * activity) - 1;
        }

        return p;
}

//-------------------------------------------------------------

bool Simulation::UpdateWeights(SimulationRecord& rec, vector<double>& firing_rates)
{
        // Record if node exceeds regulation
```

```
bool homeostatic_break[kCTRNN_NUM_NODES];
memset(homeostatic_break, 0, kCTRNN_NUM_NODES*sizeof(bool));

double double_max_weight = kMAX_WEIGHT_SIZE * 2.0;

for (int i = 0; i < kCTRNN_NUM_NODES; i++) {
        for (int j = 0; j < kCTRNN_NUM_NODES; j++) {

                double weight = rec.weights[i][j];
                double jY     = rec.values[j];
                double jBias  = rec.biases[j];
                double iN     = rec.rate[i][j];
                double iRate  = firing_rates[i];
                double jRate  = firing_rates[j];

                double jPlastic = Plasticity(4.0, 2.0, jY, jBias);

                // Check there is no plastic changes for a neuron
                if (jPlastic > 0.0) {
                        homeostatic_break[j] = true;
                }

                double delta = 0.0;

                // convert current weight to threshold, map range [-8,8] to [0, 1]
                double threshold = (weight + kMAX_WEIGHT_SIZE) / (double_max_weight);

                // Do linear damping on current weight size
                double damping = kMAX_WEIGHT_SIZE - fabs(weight);

                int rule = rec.rules[i][j];
                switch (rule)
                {
                        case 0: {
                                        // R0 rule change = delta * p * zi * zj
                                        delta = damping * iN * jPlastic * iRate * jRate;
                                }
                                break;

                        case 1: {
                                        delta = damping * iN * jPlastic * (iRate - threshold) * jRate;
                                }
                                break;

                        case 2: {
                                        delta = damping * iN * jPlastic * iRate * (jRate - threshold);
                                }
                                break;

                        case 3:
                                // do nothing - no update
                                break;
                }

                // Update the weight
                weight = weight + delta;
                if (weight < kMIN_WEIGHT_SIZE) {
                        weight = kMIN_WEIGHT_SIZE;
                }
                else if (weight > kMAX_WEIGHT_SIZE) {
                        weight = kMAX_WEIGHT_SIZE;
                }

                // Store weight back in connection array
                rec.weights[i][j] = weight;
```

```
                }
        }

        // Count nodes that didn't break homestatic regulation
        for (int i =0; i != kCTRNN_NUM_NODES; i++) {
                if (homeostatic_break[i] == false)
                        rec.homeostatic++;
        }
        return true;
}

//-------------------------------------------------------------
// Runs the CTRNN for a few steps to enable it to settle from
// random starting position
bool Simulation::PreIntegrate (SimulationRecord& rec)
{
        bool ok = true;
        vector<double> sig (kCTRNN_NUM_NODES);

        for (int step = 0; step < kNUMBER_OF_PREEVAL_STEPS; step++) {

                // Provide no input
                {
                        rec.inputs[0] = 0.0;
                        rec.inputs[1] = 0.0;
                }

                // integrate network in this time step
                for (int i= 0; i < kCTRNN_NUM_NODES; i++) {

                        // -yi
                        double change = -1.0 * rec.values[i];

                        // Now add in inputs from other nodes
                        for (int j = 0; j < kCTRNN_NUM_NODES; j++) {

                                // contribution of other nodes times their connection weight
                                float temp = rec.values[j] + rec.biases[j];

                                // Use Sigmoid lookup table
                                if (temp < -10) {
                                        sig[j] = SIGMOID[0];
                                }
                                else if (temp >= 10) {
                                        sig[j] = SIGMOID[9999];
                                }
                                else {
                                        sig[j] = SIGMOID[(int)(temp*500)+5000];
                                }
                                change += rec.weights[j][i] * sig[j];
                        }

                        // read input for node
                        change += rec.inputs[i];

                        // Apply time value of node
                        change *= (1.0 / rec.ntimes[i]);

                        // Record change in this time step
                        rec.values[i] = (float) rec.values[i] + (float)(kTIME_STEP * change);

                        // Finally if values have gone out of bound agent is failed
                        if ( isnan(rec.values[i]) ) {
```

```
                                // values are out of range
                                ok = false;
                                break;
                        }
                }
        }
        return ok;
}

//-------------------------------------------------------------

bool Simulation::Integrate (SimulationRecord& rec, int steps, bool log)
{
        float near_threshold = 4.0 * kAGENT_RADIUS;
        vector<double> sig (kCTRNN_NUM_NODES);

        float inoise1, inoise2, mnoise1, mnoise2;

        for (int step = 0; (step < steps) && (!rec.failed) ; step++) {

                // Sense light sensor
                // NOTE: First two nodes in genotype are input
                {
                        m_agent->Sense(rec.inputs, rec.source);

                        // Adjust sense values by genetic gain
#if kEVOLVE_WITH_NOISE
                        inoise1 = RandomLib::RangedGausDouble(0.0, 0.5, 1.0, 0.25);
                        inoise2 = RandomLib::RangedGausDouble(0.0, 0.5, 1.0, 0.25);
#else
                        inoise1 = inoise2 = 0.0;
#endif
                        rec.inputs[0] = (rec.inputs[0] + inoise1) * exp(rec.sensor_gain);
                        rec.inputs[1] = (rec.inputs[1] + inoise2) * exp(rec.sensor_gain);

#if DEBUG
                        float i1 = rec.inputs[0];
                        float i2 = rec.inputs[1];
#endif
                        // Peform Sensory inversion
                        if (rec.invert == true) {
                                float temp = rec.inputs[0];
                                rec.inputs[0] = rec.inputs[1];
                                rec.inputs[1] = temp;
                        }

#if DEBUG
                        assert(!isnan(rec.inputs[0]));
                        assert(!isnan(rec.inputs[1]));
#endif
                }

                // integrate network in this time step
                for (int i= 0; i < kCTRNN_NUM_NODES; i++) {

                        // -yi
                        double change = -1.0 * rec.values[i];

                        // Now add in inputs from other nodes
                        for (int j = 0; j < kCTRNN_NUM_NODES; j++) {

                                // contribution of other nodes times their connection weight
                                float temp = rec.values[j] + rec.biases[j];

                                // Use Sigmoid lookup table
```

```
                        if (temp < -10) {
                                sig[j] = SIGMOID[0];
                        }
                        else if (temp >= 10) {
                                sig[j] = SIGMOID[9999];
                        }
                        else {
                                sig[j] = SIGMOID[(int)(temp*500)+5000];
                        }
                        change += rec.weights[j][i] * sig[j];
                }

                // read input for node
                change += rec.inputs[i];

                // Apply time value of node
                change *= (1.0 / rec.ntimes[i]);

                // Record change in this time step
                rec.values[i] = (float) rec.values[i] + ((float) kTIME_STEP * (float) change);

                // Finally if values have gone out of bound agent is failed
                if ( isnan(rec.values[i]) ) {

                        // values are out of range
                        rec.failed = true;
                        break;
                }
        }

        // Perform plasticity on each synapse
#if kALLOW_PLASTIC_CHANGES
        UpdateWeights(rec, sig);
#endif

        // generate motor values
        // NOTE: Last two nodes in genotype are motor effectors
        if (!rec.failed)
        {
                assert(kCTRNN_NUM_NODES >= 4);

                // Get firing rate for neurons
                int lmotor = kCTRNN_NUM_NODES - 1;
                int rmotor = kCTRNN_NUM_NODES - 2;
                float vr = Sigmoid(rec.values[lmotor] + rec.biases[lmotor]);
                float vl = Sigmoid(rec.values[rmotor] + rec.biases[rmotor]);

                // map input to range [-1, 1] - current range is [0, 1]
                vr = (float) (vr * 2.0) - 1.0f;
                vl = (float) (vl * 2.0) - 1.0f;

                // Adjust motor values by genetic gain
#if kEVOLVE_WITH_NOISE
                mnoise1 = (float) RandomLib::RangedGausDouble(-0.25, 0.25, 1.0, 0.0);
                mnoise2 = (float)RandomLib::RangedGausDouble(-0.25, 0.25, 1.0, 0.0);
#else
                mnoise1 = mnoise2 = 0.0;
#endif
                vr    = (vr + mnoise1) * rec.motor_gain;
                vl    = (vl + mnoise2) * rec.motor_gain;

                // Get agent to update its position based on motor values
                m_agent->RunOneUnit(vr, vl, kTIME_STEP);

                // See if agent is near light source
```

```
                        Position current_pos = m_agent->Pos();
                        Position light_pos   = rec.source->Pos();
                        float distance = DistanceBetweenPoints(light_pos.x, light_pos.y, \
                                                                                current_pos.x, current_pos.y);

                        if (log) {
                                DistanceLog(rec, distance);
                        }

                        if (distance < near_threshold) {
                                rec.near ++;
                        }
                }

                // See if we should log
                if (log)  {
                        ActivityLog(rec, sig);
                        WeightsLog(rec);
                }

                rec.step_counter++;
                rec.total_steps++;
        }
        return (!rec.failed);
}

//------------------------------------------------------
// Lights all placed near agent starting position
void Simulation::GenerateLights (Position pos)
{
        m_sources.clear();

        LightSource * source = 0;
        for (int n = 0; n < kNUMBER_LIGHTS_PER_PRESENTATION; n++) {
                source = new LightSource(pos);
                m_sources.push_back(source);
        }
}

//------------------------------------------------------

void Simulation::ClearLights ()
{
        while (!m_sources.empty()) {
                LightSource *s_p = m_sources.back();
                m_sources.pop_back();
                delete s_p;
        }
}


//------------------------------------------------------

bool Simulation::RunMicrobial ()
{

        m_buffer = "Evolving";

        Genotype* aGene, *bGene, *pWinner, *pLoser;
        int   elite = 0;                        // index of 'best' current population member
        float elite_fitness = 0.0;

        // Fitness value of each population member
        vector<float> fitness           (m_pop, 0.0);
```

```
        // Average fitness value over length of GA
        float avg_fitness  [kNUMBER_OF_GENERATIONS];
        float worst_fitness[kNUMBER_OF_GENERATIONS];
        float best_fitness [kNUMBER_OF_GENERATIONS];

        char buffer[512];
        // Actual step counter
        int count = 0;
        int gen_count = 0;
        int winner_idx = 0;;

        for (int i = 0; i < kNUMBER_OF_GENERATIONS; i ++, gen_count++) {
                for (int n = 0; n < kPOPULATION_SIZE; n++, count++) {
                        printf("[%d]", count);

                        // pick 2 at random
                        int a = static_cast<int>(kPOPULATION_SIZE * RandomLib::Double());
                        int b = static_cast<int>(kPOPULATION_SIZE * RandomLib::Double());

                        // Try again - but doesn't matter if they're the same
                        if ( a == b)
                                 b = static_cast<int>(kPOPULATION_SIZE * RandomLib::Double());

                        // Look for fitess genotype
                        aGene = (*m_genes)[a];
                        bGene = (*m_genes)[b];

                        fitness[a] = (float) EvaluateGenotype(aGene);
                        fitness[b] = (float) EvaluateGenotype(bGene);

                        if (fitness[a] > fitness[b]) {
                                pWinner = aGene; pLoser = bGene;
                                winner_idx = a;
                        }
                        else {
                                pWinner = bGene; pLoser = aGene;
                                winner_idx = b;
                        }

                        if (fitness[winner_idx] > elite_fitness) {
                                elite_fitness = fitness[winner_idx];
                                elite         = winner_idx;

                                // Record elite agent for testing
                                SaveElite(elite);
                        }

                        MutateGenotype(*pWinner, *pLoser);                                              //end of GA part
                }

                // ---- Record generational stats
                worst_fitness[gen_count] = *min_element(fitness.begin(), fitness.end()) ;
                best_fitness[gen_count]  = *max_element(fitness.begin(), fitness.end()) ;
                avg_fitness[gen_count]   = FloatVectorMean(fitness, m_pop);

                // ---- Display generation information
#ifdef WIN_ENV
                _snprintf_s(buffer, 512, "Fitest = %d (%.2f), Avg. = %.2f - step (%d of %d)", elite, elite_fitness, \
                                                                        avg_fitness[gen_count], count, kNUMBER_OF_GENERA

                // Callback any registered function
                if (m_prog) {

                        string eliteStr ;
                        Genotype * pElite = (*m_genes)[elite];
```

```
                        if (pElite) {
                                eliteStr = GenotypeToString(*pElite);
                        }
                        (m_prog)(buffer, eliteStr.c_str(), m_data);
                }
#else
                snprintf(buffer, 512, "Fitest = %d (%.2f), Avg. = %.2f - step (%d of %d)", elite, elite_fitness, \
                                                                                        avg_fitness[gen_count], count, kNUMBER_OF_GENERA
                PrintVector (fitness, buffer);
#endif
        }

        //--- Save Data
        {
                // Save Fitest members of the population
                FloatVector rank = fitness;

                // Sort fitness into ascending order
                std::sort(rank.begin(), rank.end());

                // Chose the fitness to pick approx 10 fitest
                int threshold = rank[m_pop - 10];

                // Now loop and save fitest agents
                stringstream fileName;

                int index = 0;
                int fitest = 0;

                for (int n = 0; n < m_pop; n++) {
                        if (fitness[n] >= threshold) {
                                // Save population member
                                fileName << kDEFAULT_FILE_PATH;
                                fileName << n;
                                fileName << kDEFAULT_END_AGENT_FILE;
                                SavePopulationMember(fileName.str(), n);

                                fileName.str("");               // clear
                                fileName << kDEFAULT_FILE_PATH;
                                fileName << n;
                                fileName << "EndAgent.dot";
                                ExportPopulationMemberAsDot(fileName.str(), n);
                        }

                        if (fitness[n] > fitest) {
                                index = n;
                                fitest = fitness[n];
                        }
                }

                //---- Save Fitest ------
                // Save population member in default file
                fileName.str("");               // clear
                fileName << kDEFAULT_FILE_PATH;
                fileName << kDEFAULT_END_AGENT_FILE;
                SavePopulationMember(fileName.str(), index);

                fileName.str("");               // clear
                fileName << kDEFAULT_FILE_PATH;
                fileName << "EndAgent.dot";
                ExportPopulationMemberAsDot(fileName.str(), index);

                //---- Dump Statistics
                fileName.str("");               // clear
                fileName << kDEFAULT_FILE_PATH;
```

```
                fileName << kDEFAULT_FITNESS_FILE;
                DumpFitness(worst_fitness, avg_fitness, best_fitness, kNUMBER_OF_GENERATIONS, \
                                        fileName.str().c_str(), "Fitness");
        }

        ClearPopulation(*m_genes);

        m_buffer = "done";
        return true;
}

//------------------------------------------------------

void Simulation::SaveElite (int elite)
{
        if ((elite < 0) || (elite >= m_pop))
                throw std::logic_error("Population index out of range");

        string fileName = kDEFAULT_FILE_PATH;
        fileName += "EliteAgent.dot";
        ExportPopulationMemberAsDot(fileName, elite);

        // Save Elite member
        fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_CHAMPION_AGENT_FILE;
        SavePopulationMember(fileName, elite);
}

//------------------------------------------------------
// Runs the simulation with rank based selection
bool Simulation::RunRank ()
{
        Genotype* aGene, *pWinner, *pLoser;
        int elite;
        float fitest = 0.0;

        // Fitness value of each population per cycle
        vector<float> fitness (m_pop);

        // Average fitness value over length of GA
        float avg_fitness [kNUMBER_OF_GENERATIONS];

        char buffer[512];

        for (int n = 0; n < kNUMBER_OF_GENERATIONS; n++)
        {
                // Start where agent starts
                Position start_pos = {0.0, 0.0};
                GenerateLights(start_pos);

                for (int i = 0; i < m_pop; i++) {
                        aGene = (*m_genes)[i];
                        fitness[i] = (float) EvaluateGenotype(aGene);
                }


                // Now do rank selection
                multimap<float, int> order;
                for (int i = 0; i < m_pop; i ++) {
                        order.insert(std::pair<float, int>(fitness[i], i));
                }

                // Keys will be ordered in ascending fitness
                multimap<float, int>::reverse_iterator  rpos;
                multimap<float, int>::iterator                  fpos;
```

```
                // Remember elitism - preseve best member of previous generation
                // now seed new generation
                int desired = static_cast<int>(m_pop / 3.0f);

                rpos = order.rbegin();
                fpos = order.begin();

                // First copy the 'elite' group - so now have a population
                // With 2 copies of each
                for (int i=0; i < desired; i++, ++fpos, ++rpos) {
                        // Replace a 'loser' with one from the elite group
                        int loser = fpos->second;
                        int winner = rpos->second;

                        pWinner = (*m_genes)[winner];
                        pLoser  = (*m_genes)[loser];
                        MutateGenotype(*pWinner, *pLoser);
                }

                // Return to the start of the elite group
                rpos = order.rbegin();
                elite = rpos->second;
                ++rpos;         // skip best of previous generation

                // Now mutate all the other 'original copies'
                desired *= 2;
                for (int i = 1; i < desired; i++, ++rpos) {
                        int winner = rpos->second;

                        pWinner = (*m_genes)[winner];

                        MutateGenotype(*pWinner);
                }

                // ---- Dump stats
                avg_fitness[n] = FloatVectorMean(fitness, m_pop);

#if WIN_ENV
                _snprintf_s(buffer, 512, "Fitest = %d (%.2f), Avg. = %.2f - step (%d of %d)", elite, fitness[elite], \
                                                                                avg_fitness[n], n, kNUMBER_OF_GENERATIONS);
#else
                // Display generation information
                snprintf(buffer, 512, "\nFitest = %d (%.2f), Avg. = %.2f - step (%d of %d)", elite, fitness[elite], \
                                                                                avg_fitness[n], n, kNUMBER_OF_GENERATIONS);
#endif

                PrintVector (fitness, buffer);

                if (fitness[elite] == fitest) {

                        // Save 'elite' population member
                        string fileName = kDEFAULT_FILE_PATH;
                        fileName += kDEFAULT_CHAMPION_AGENT_FILE;
                        SavePopulationMember(fileName, elite);
                        fitest = fitness[elite];
                }

                // Clear out light array
                ClearLights();
        }

        //--- Save Fitess member of the population
        {
                float fitest = 0.0;
```

```
                int index = 0;

                for (int n = 0; n < m_pop; n++) {
                        if (fitness[n] > fitest) {
                                fitest = fitness[n];
                                index = n;
                        }
                }

                // Save population member
                string fileName = kDEFAULT_FILE_PATH;
                fileName += kDEFAULT_END_AGENT_FILE;
                SavePopulationMember(fileName, index);

                //---- Dump Statistics
                fileName = kDEFAULT_FILE_PATH;
                fileName += kDEFAULT_FITNESS_FILE;
                DumpFitness(avg_fitness, NULL, NULL, kNUMBER_OF_GENERATIONS, fileName, "Fitness");
        }

        ClearPopulation(*m_genes);

        return false;
}

//-----------------------------------------------------

bool Simulation::StepInit (std::string fileName, int number_of_trials,
                                                        bool invert, bool stationary)
{
        Genotype* gene = LoadPopulationMember(fileName);

        if (m_rec) {
                delete m_rec;
        }

        m_rec = new SimulationRecord();
        InitSimulationRecord(*m_rec, *gene);

        if (stationary)
                m_rec->motor_gain = 0.0;

        // clear gene
        delete gene;

        // Clear any old information
        delete m_agent;

        Position start_pos = {0.0, 0.0};
        float   max_angle = 360.0;
        if (stationary) {
                max_angle = 0.0;                // agent points straight
        }
        m_agent = new Agent(start_pos, kAGENT_RADIUS, \
                                        static_cast<float>(RandomLib::Double() * max_angle));

        // Initialise record
        m_rec->source       = new LightSource(start_pos, max_angle);
        m_rec->source_total = number_of_trials;
        m_rec->source_count = 0;

        // Invert step point
        if (invert) {
                m_rec->invert_step  = (int) (number_of_trials/4);
        }
```

```
        StartLogs();

        return true;
}

//-----------------------------------------------------

float Simulation::CalcFitness (SimulationRecord& rec, float& d, float&n, float& h)
{
        float fitness = 0.0;
        d = n = h = 0.0;

        if (!rec.failed) {
                int steps = rec.source->Time() / kTIME_STEP;

                Position start_pos   = m_agent->StartPos();
                Position light_pos   = rec.source->Pos();
                float start_distance = DistanceBetweenPoints(light_pos.x, light_pos.y, start_pos.x, start_pos.y);

                Position end_pos = m_agent->Pos();
                float end_distance = DistanceBetweenPoints(light_pos.x, light_pos.y, end_pos.x, end_pos.y);

                // has agent moved nearer the light?
                d = (float) (1.0 - (end_distance / start_distance));
                if (d < 0.0) {
                        d = 0.0;
                }

                // Proportion of time spent near the light
                n = rec.near / (float) steps;

                // Time average number of neurons behaving homeostatically
#if kALLOW_PLASTIC_CHANGES
                h = (rec.homeostatic) / ((float) steps * kCTRNN_NUM_NODES);
#else
                h = 0.0;
#endif

                // Average of end point and time spent at the light source
                fitness = (kALPHA_MODIFIER * d) + (kBETA_MODIFIER * n) + (kGAMMA_MODIFIER * h);
        }
        return fitness;
}

//-----------------------------------------------------
// Called by timer callback - decided where in the simulation
// we are.

bool Simulation::StepIntegrate ()
{
        if (m_rec && m_rec->done) {
                return true;
        }

        // Do five steps
        Integrate(*m_rec, 1, true);

        // See if we need a new light source
        if (m_rec->step_counter > (m_rec->source->Time() / kTIME_STEP)) {

                float d, h, n;
                float f = CalcFitness(*m_rec, d, n, h);
                FitnessLog(*m_rec, f, d, n, h);

                // Check if we need a new light source
```

```
                if (m_rec->source_count < m_rec->source_total) {
                        Position pos = m_agent->Pos();
                        m_rec->source = new LightSource(pos);
                        m_rec->step_counter = 0;
                        m_rec->source_count++;
                        m_rec->near        = 0.0;
                        m_rec->homeostatic = 0.0;

                        // Perform sensory inversion if needed
                        // Note: cannot invert on the first run
                        if ( (m_rec->invert_step != 0) && (m_rec->invert_step <= m_rec->source_count) ) {
                                m_rec->invert = true;
                        }

                        std::stringstream buf;
                        buf << "Lightsource " << m_rec->source_count << " of " << m_rec->source_total << " Inversion: " << m_rec
                        m_buffer = buf.str();
                }
                else {
                        // Signal that run is over
                        m_rec->done = true;
                        m_buffer = "done";
                        EndLogs();
                }
        }

        return true;
}


//-------------------------------------------------------------
// Log & Export Code
//-------------------------------------------------------------

//-------------------------------------------------------------

void Simulation::Draw()
{
        if (m_agent != NULL) {
                m_agent->Draw();
        }

        if ((m_rec != NULL) && (m_rec->source != NULL)) {
                m_rec->source->Draw();
        }

        if (!m_buffer.empty()) {
                DrawText(m_buffer.c_str(), 0.0, 0.0, 8);
        }
}

//-------------------------------------------------------------
// Logs information about an agent's state
void Simulation::ActivityLog (SimulationRecord& rec, DoubleVector& firing_rates)
{
        if (aLog.is_open()) {
                aLog << rec.total_steps << "\t";
                aLog << rec.inputs[0] << "\t";
                aLog << rec.inputs[1] << "\t";

                DoubleVector::iterator dpos = firing_rates.begin();
                for ( ; dpos != firing_rates.end(); ++dpos) {
                        aLog << (*dpos) << "\t";
                }
```

```
                FloatVector::iterator fpos = rec.values.begin();
                for ( ; fpos != rec.values.end(); ++fpos) {
                        aLog << (*fpos) << "\t";
                }

                aLog << endl;
        }
}

//--------------------------------------------------------
// Logs information about an agent's state
void Simulation::DistanceLog (SimulationRecord& rec, float distance)
{
        if (dLog.is_open()) {
                dLog << rec.total_steps << "\t" << distance << endl;
        }
}

//--------------------------------------------------------
// Logs information about an agent's state
void Simulation::FitnessLog (SimulationRecord& rec, float overall, float distance, float near, float homeo)
{
        if (fLog.is_open()) {
                fLog << rec.source_count << "\t" << overall << "\t";
                fLog << distance << "\t" << near << "\t" << homeo << endl;
        }
}

//--------------------------------------------------------
// Logs information about an agent's state
void Simulation::WeightsLog (SimulationRecord& rec)
{
        if (wLog.is_open()) {
                wLog << rec.total_steps << "\t";

                for (int i = 0; i < kCTRNN_NUM_NODES; i++) {
                        for (int j = 0; j < kCTRNN_NUM_NODES; j++) {
                                wLog << RescaleUnitValue(kMIN_WEIGHT_SIZE, kMAX_WEIGHT_SIZE, rec.weights[i][j]) << "\t";
                        }
                }
                wLog << endl;
        }
}

//--------------------------------------------------------

void Simulation::StartLogs()
{
        EndLogs();

        std::string fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_ACTIVITY_LOG;
        aLog.open(fileName.c_str(), ios::out);
        aLog << "#plot \"" << fileName << "\"";
        aLog << " using 1:2 title \'plot-title\' with lines" << std::endl;
        aLog << "#step_count\tinput1\tinput2\tfiring rates [1-" << kCTRNN_NUM_NODES << "]";
        aLog << "\tCell Potentials [1-" << kCTRNN_NUM_NODES <<"]" << endl;

        fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_DISTANCE_LOG;

        dLog.open(fileName.c_str(), ios::out);
        dLog << "#plot \"" << fileName << "\"";
        dLog << " using 1:2 title \'plot-title\' with lines" << std::endl;
        dLog << "#step\tdistance" << endl;
```

```
        fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_FITNESS_LOG;

        fLog.open(fileName.c_str(), ios::out);
        fLog << "#plot \"" << fileName << "\"";
        fLog << " using 1:2 title \'plot-title\' with lines" << std::endl;
        fLog << "#step\toverall\tdistance\tnear\thomeo" << endl;

        fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_WEIGHT_LOG;

        wLog.open(fileName.c_str(), ios::out);
        wLog << "#plot \"" << fileName << "\"";
        wLog << " using 1:2 title \'plot-title\' with lines" << std::endl;
        wLog << "#step\t[weights for each neuron 0 - 4]" << endl;
}

//---------------------------------------------------------

void Simulation::EndLogs()
{
        aLog.close();
        dLog.close();
        fLog.close();
        wLog.close();
}

//---------------------------------------------------------

void Simulation::SavePopulation (std::string fileName)
{

}

//---------------------------------------------------------------

void Simulation::ExportPopulationMemberAsDot (std::string fileName, int index)
{
        if ((index < 0) || (index >= m_pop))
                throw std::logic_error("Population index out of range");

        ofstream out;
        out.open(fileName.c_str(), ios::out);

        Genotype* aGene = (*m_genes)[index];

        if (aGene) {
                std::string outStr = GenotypeToDot(*aGene);
                out << outStr;
        }

        out.close();
}

//---------------------------------------------------------------

void Simulation::SavePopulationMember (std::string fileName, int index)
{
        if ((index < 0) || (index >= m_pop))
                throw std::logic_error("Population index out of range");

        ofstream out;
        out.open(fileName.c_str(), ios::out);
```

```
        Genotype* aGene = (*m_genes)[index];

        RealGenotype::iterator pos = aGene->rgene.begin();
        for (; pos != aGene->rgene.end(); ++pos) {
                out << (*pos) << " ";
        }

        IntGenotype::iterator ipos = aGene->igene.begin();
        for (; ipos != aGene->igene.end(); ++ipos) {
                out << (*ipos) << " ";
        }

#if DEBUG
        printf("\nSaved genotype (%s):\n", fileName.c_str());
        PrintGenotype(*aGene);
#endif

        out.close();
}

//--------------------------------------------------------------

Genotype* Simulation::LoadPopulationMember (std::string fileName)
{
        ifstream in;
        in.open(fileName.c_str());

        assert(in.is_open());

        Genotype* gene = new Genotype();

        // Read whole file as vectors
        double val = 0;
        for (int i= 0; i < kREAL_GENOTYPE_LENGTH; i++){
                in >> val;
                gene->rgene.push_back(val);
        }

        int ival = 0;
        for (int i= 0; i < kINT_GENOTYPE_LENGTH; i++){
                in >> ival;
                gene->igene.push_back(ival);
        }

#if DEBUG
        printf("\nLoaded genotype (%s):\n", fileName.c_str());
        PrintGenotype(*gene);
#endif
        return gene;
}

//--------------------------------------------------------------

void DumpFitness (float* vect1, float* vect2, float* vect3, int length,
                                std::string fileName, const char* title)
{
        std::ofstream fit_file;
        fit_file.open(fileName.c_str());

        fit_file << "#plot \"" << fileName << "\"";
        fit_file << " using 1:2 title '" << title << "' with lines" << std::endl;
        fit_file << "#Worst\tAvg\tBest" << endl;

        FloatVector::iterator pos;
        for (int i = 0; i < length; i++) {
```

```
                fit_file << i << "\t" << vect1[i];

                if (vect2)
                        fit_file << "\t" << vect2[i];

                if (vect3)
                        fit_file << "\t" << vect3[i];
                fit_file << std::endl;
        }
        fit_file.close();
}

#if WIN_ENV

//--------------------------------------------------------------

void Simulation::AddProgCB(ProgCall prog, void * data)
{
        m_prog = prog;
        m_data = data;
}

#endif



//--------------------------------------------------------------
// Local Helper Functions
//--------------------------------------------------------------


//--------------------------------------------------------------
// Print Fitness
void PrintVector (vector<float>& fit, const char* title)
{
        printf("%s\n", title);
        vector<float>::iterator pos;
        for (pos = fit.begin(); pos != fit.end(); ++pos) {
                float value = (*pos);
                printf("%.2f\t", value);
        }
        printf("\n");

}


//--------------------------------------------------------------
// TODO: replace with a lookup table. Its much much faster
float Sigmoid (float x)
{
        float divisor = (1.0f + exp(-x));
        return (1.0f / divisor);
}


//--------------------------------------------------------------

float FloatVectorMean (vector<float>& input, int n)
{
        float mean = 0.0;

        vector<float>::iterator pos;
        for (pos = input.begin(); pos != input.end(); ++pos) {
                mean += (*pos);
        }
```

```
        mean /= (float) n;
        return mean;
}


//-------------------------------------------------------------

float FloatVectorStdDev (vector<float>& input, float mean, int n)
{
        float sum = 0.0;

        vector<float>::iterator pos;
        for (pos = input.begin(); pos != input.end(); ++pos) {
                sum += pow((*pos - mean), 2);
        }
        sum /= (float) n;
        return sum;
}
```

SimulationRecord.h

```
/*
 *  SimulationRecord.h
 */
#pragma once

#include <vector>

typedef std::vector<float>  FloatVector;
typedef std::vector<double> DoubleVector;

class LightSource;              // forward ref

/**
 * Simulation record object, keeps track of state during simulation
 */
typedef struct
{
        //---- Genotype information -----------------
        FloatVector             ntimes;                                         //!< time values of node
        FloatVector             biases;                                         //!< bias values for nodes
        FloatVector             values;                                         //!< current value for nodes
        FloatVector             inputs;                                         //!< current input for nodes
        double weights[kCTRNN_NUM_NODES][kCTRNN_NUM_NODES];     // weights
        double rate   [kCTRNN_NUM_NODES][kCTRNN_NUM_NODES];     // learning rate
        int        rules  [kCTRNN_NUM_NODES][kCTRNN_NUM_NODES]; // learning rules
        float                   motor_gain;                     //!< Motor gain for the genotype
        float                   sensor_gain;            //!< Input sensor gain for the genotype

        //---- Trial information --------------------
        LightSource*    source;                         //!< Current light source
        int                             source_count;           //!< Number of light sources created.
        int                             source_total;           //!< Total number of sources to create

        int                             step_counter;           //!< integration counter
        int                             total_steps;            //!< total_steps performed during integration
        int                             near;                           //!< Counter for agent evaluation - number of steps near
        int                             homeostatic;            //!< Counter for agent evaluation - number of neurons behaving h
        bool                    failed;                         //!< will be set to true if agent does something catastrophic
        bool                    done;                           //!< Simulation has ended

        int                             invert_step;            //!< Light source number to invert at
        bool                    invert;                         //!< If true, invert the sensors
}
SimulationRecord;
```

SimulationRecord.cpp

```
/*
 *  SimulationRecord.cpp
 */

#include <vector>
using namespace std;

#include "globals.h"
#include "Genetics.h"
#include "Random.h"
#include "SimulationRecord.h"
#include "Geo.h"

//---------------------------------------------------------------
// Reserve space in rec and load the genotype into a more
// convenient form
void InitSimulationRecord (SimulationRecord& rec, Genotype& gene)
{
        rec.ntimes.reserve(kCTRNN_NUM_NODES);
        rec.biases.reserve(kCTRNN_NUM_NODES);
        rec.values.reserve(kCTRNN_NUM_NODES);
        rec.inputs.reserve(kCTRNN_NUM_NODES);

        // Read in genotype into more convenient data structure
        RealGenotype::iterator pos = gene.rgene.begin();
        float gain = (float)(*pos);
        rec.motor_gain = RescaleUnitValue(kMIN_MOTOR_GAIN, kMAX_MOTOR_GAIN, gain);      // record motor gain
        pos++;

        gain = (float) (*pos);                                  // record sensor gain
        rec.sensor_gain = RescaleUnitValue(kMIN_INPUT_GAIN, kMAX_INPUT_GAIN, gain);     // record motor gain
        pos++;

        float bias, time, weight, rate;

        for (int i = 0; i < kCTRNN_NUM_NODES; i++) {
                // load node time value
                time = (float) (*pos);
                time = RescaleUnitValue(kMIN_TIME_SIZE, kMAX_TIME_SIZE, time);
                rec.ntimes.push_back(time);                     // record time for the node
                ++pos;

                // load node bias value
                bias = (float) (*pos);
                bias = RescaleUnitValue(kMIN_BIAS_SIZE, kMAX_BIAS_SIZE, bias);
                rec.biases.push_back(bias);                     // record bias for the node
                ++pos;

                // load synaptic weights into matrix
                for (int j = 0; j < kCTRNN_NUM_NODES; j++) {
                        if (pos != gene.rgene.end()) {
                                weight = (float) (*pos);
                                weight = RescaleUnitValue(kMIN_WEIGHT_SIZE, kMAX_WEIGHT_SIZE, weight);
                                rec.weights[i][j] = weight;
                        }
                        ++pos;
                }

                // load synaptic learning rate into matrix
                for (int j = 0; j < kCTRNN_NUM_NODES; j++) {
                        if (pos != gene.rgene.end()) {
                                rate = (float) (*pos);
                                rate = RescaleUnitValue(kMIN_RATE_OF_CHANGE, kMAX_RATE_OF_CHANGE, rate);
```

```
                        rec.rate[i][j] = rate;
                }
                ++pos;
        }

        // Fill values with starting points
        // randomise the starting positions of the network to range [-bias-1,-bias+1]
        // Bias trick is suggested by Eduardo
        rec.values.push_back((float)RandomLib::RangedDouble(-bias - 1.0, -bias + 1.0));
        rec.inputs.push_back(0.0);
}

// Copy learning rules into matrix
IntGenotype::iterator ipos = gene.igene.begin();
for (int i = 0; i < kCTRNN_NUM_NODES; i++) {
        for (int j = 0; j < kCTRNN_NUM_NODES; j++, ++ipos) {
                rec.rules[i][j] = (*ipos);
        }
}

// clear counters
rec.step_counter = 0;
rec.source_count = 0;
rec.total_steps  = 0;

// Start agent in non-failed state
rec.done        = false;
rec.failed      = false;
rec.near        = 0;
rec.homeostatic = 0;
rec.source      = 0;
rec.source_total= 0;
rec.invert              = false;
rec.invert_step = 0;
}
```