On Dynamics of Searching Behaviours

Richard Potter

March 5, 2007

1 Introduction

Nepomnyashchikh & Podgronyj [13] proposed that searching behaviour in organisms may not have been developed by natural selection but may have been 'ready-made' in the properties of non-dynamical systems.

Because the field of Artificial Life can be seen as the attempt to synthesize lifelike behaviour, their work has much that is compelling. It shows how behaviour may emerge from the substrate of an agent alone; no explicit controller is required. Such an agent could be seen as an extension of the work of Braitenberg [6], where rich observable behaviour emerges from understandable components with no centre of control.

Nepomnyashchikh & Podgronyj are guided by their view that 'all organisms, from those that are unicellular to higher animals, are capable of displaying spontaneous behaviour' and use examples from bacterial, mice and fruit fly studies to show that central pattern generators (CPG) may drive spontaneous activity in these organisms.

The ability to be self-mobile is considered by many to be an important facilitator for cognition [9] [16], enabling an agent to potentially form a sense of self and give rise to intelligence. An organism has to expend energy to maintain its identity as a mobile unit in space [16]; therefore it must be able to take in energy from its environment. For most biological systems energy must be located through searching for uncertain targets in a manner that avoids revisiting previously searched areas.

Such spontaneous activity can also be seen as a method of forcing pertubations in the system, for example causing optic-flow information to be generated enabling the agent to interact, explore and 'play' in its environment. Mobus et al [12] required their agent to be able to locate distributed resources in the environment. They noted that the search strategy used has 'a significant impact on how quickly [the agent] is exposed to instances of successfully finding a resource and consequently learning'.

Related work has been performed by Freeman [8] who showed the central olfactory system can be modelled by using similar non-linear components, whose behaviour simulates the chaotic patterns of the Electroencephalograms (EEG). Parallel work with central pattern generators has been performed with robots [7], showing that non-linear dynamics underpin many biological and biologically inspired processes.

In repeating Nepomnyashchikh & Podgronyj's study, the main issues to explore are:

1. What is the simplest agent capable of generating searching rules? In their study the authors state that it was not their intent to find the least complicated possible system capable of generating search behaviours.
2. To what extent are the searching rules that appear dependent on the type of oscillator? Are there simple biological oscillators that exhibit interesting behaviours? Or is plausible searching dependent on the rich dynamics logistic maps?
3. How can wandering behaviour be defined as biologically plausible? Or is it enough that its not biologically implausible?

2 Nepomnyashchikh & Podgronyj's Agent and Model

Nepomnyashchikh & Podgronyj proposed an agent consisting of three oscillators and a single odour sensor (Figure 1). The oscillators are always active, but they could be inhibited by input from the sensor. The activating oscillator influences the activity in the left and right turn oscillators via excitatory connections. The left and right turn oscillators inhibit each other. The sensory input inhibits then both by sampling the increment of ambient stimulation in the simulated environment.

Each oscillator uses a non-linear logistic map (Section 2.1), with white Gaussian noise added to each oscillator to simulate internal noise of the system.

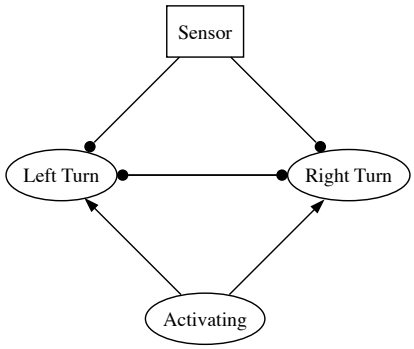

Figure 1: **Architecture of the Agent.** Arrowheads represent excitatory connections & circles inhibitory connections.

The agent 'walks' simply by measuring the difference in right and left turn tendencies at a given time point i, which gives a direction in radians (1) and the agent then moves one distance unit in this direction.

$$T_i = R_i - L_i \tag{1}$$

The agent is then observed engaging in the following 'biologically inspired' tasks: wandering, orientation in odour gradients, escaping gaps in the odour gradient, area restricted searching and sampling food patches.

In the remainder of this study, this agent is referred to as the 'Logistic-map agent'.

2.1 Logistic maps as oscillators

Nepomnyashchikh & Podgronyj use the one dimensional logistic map as the basis for their 'simple non-linear dynamics'. This map has the general form of (2), a first-order nonlinear difference equation.

$$X_{i+1} = \lambda X_i(1 - X_i) \tag{2}$$

which can be expressed as:

$$X_{i+1} = \lambda F(X_i) \tag{3}$$

where $F(X_i)$ is written as:

$$F(X) = X_i(1 - X_i) \tag{4}$$

The mathematician Robert May investigated the dynamics of the logistic map [11] and noted many of the properties of (2) including:

1. X must remain between 1 and 0 otherwise subsequent iterations head towards $-\infty$.
2. $F(X)$ reaches a maximum value of $\frac{\lambda}{4}$ when $X = \frac{1}{2}$, thus the equation only has interesting dynamic behaviour when $\lambda < 4$.
3. All trajectories are attracted to $X = 0$ if $\lambda < 1$, thus for dynamic behaviour $1 < \lambda < 4$
4. The fixed point of the system becomes unstable when when $\lambda = 3$ and the 'chaotic region lies between $3.45 < \lambda < 4$.

The λ function

Each oscillator has its own λ function (7), where A_i, L_i, R_i are the values of the activating, left and right turn oscillatosr at step i, μ is the average parameter, ξ is Gaussian noise between -1.0 & 1.0 and σ is the peak magnitude of internal noise.

$$\lambda_i^A = \mu + \sigma\xi_i^A \tag{5}$$

$$\lambda_i^R = \mu + \sigma\xi_i^R + A_i - L_i - I_i \tag{6}$$

$$\lambda_i^L = \mu + \sigma\xi_i^L + A_i - R_i - I_i \tag{7}$$

For most tasks in [13], μ and σ are fixed at 1.1, to prevent the oscillators tending to zero (if $\lambda < 1$) and to hold the output in the range that is dynamically interesting. None of the oscillator outputs are allowed to become zero, instead their minimum value is fixed at 10^{-6}.

In addition the following conditions also apply $0 < A, L, R < 1$ and $0 < \lambda < 4$.

3 Alternative Agents

To understand the results of this work, the first two tasks of Nepomnyashchikh & Podgronyj's study, wandering and orientation, are repeated. This is to gain understanding into nature of the interactions of the oscillators and to enable exploration of whether the search behaviour is emergent due to the properties of the non-linear systems or has been designed in. The orientation task is especially important, as the agent is coupled with an environment (in this case via an odour gradient).

All biological searching patterns can be seen as involving a discrete series of displacement events (movement in a direction) separated by a series of orientation events (turning to a direction). For comparison two alternative agents involving different displacement and orientation techniques are used: a random walker and a sine-wave agent based on the Nepomnyashchikh & Podgronyj agent.

Random Agent

A random walk agent is used as control, because it is the simplest searching agent available. Nepomnyashchikh & Podgronyj do not compare their agent against such a walker, instead they rely on measuring diffusion of the agents in the environment as a measure of biological plausibility. If search behaviour is truly emergent, then search behaviour has to perform better than a random walk.

The random agent used in this study consists of two independent uniformly distributed pseudo random number generators. Each random number generator utilises the Mersenne Twister algorithm generating values in the range 0.0 to 1.0. One random number generator is used as a left-turn oscillator, the other as the right-turn oscillator. There are no connections between the oscillators.

The agents motion in each time step is calculated as in Section 2.

Sine Agent

A sine-wave based agent is used to determine if simple coupled oscillators can give rise to the behaviour described by Nepomnyashchikh & Podgronyj. A sine-wave oscillator replaces the logistic-map oscillators in Section 2. Each sine-wave oscillator has its own independent amplitude, frequency and phase. As in Nepomnyashchikh & Podgronyj's model, Gaussian white noise is added to oscillator. The inhibition and excitation connections are not changed.

Exchanging the logistic map oscillators with signal based oscillators, allows exploration of whether simpler agents can be constructed that exhibit searching behaviours and to partially investigate to what extent the rules depend on the type of oscillator. Sine-wave based oscillators have been used successfully in [12] resulting in agent behaviour that 'resembles foraging search [in animals] in a number of details'.

The Sine agent's values for the amplitude, frequency and phase were set by randomly filling these parameters for 1000 agents and simulating their motion for 1000 steps. The amplitude, frequency and phase settings of the agent that went the furthest from the origin was used as the settings for the Sine agent.

In addition the following conditions also apply: $0 < A < 1.0$ and $-1.0 < L, R < 1$.

Implementation Notes

The model was writen in C++, using OpenGL for display [1]. Sections of the code include the independent JPEG group library [10], Wagner's Mersenne Twister C++ class [19] and maths functions from Numerical Recipes in C [14].

4 Wandering

The first task examines how, in the absence of any such sensory stimuli, should an organism choose to explore an uncertain environment. A further question is how can the movement of agents be quantitively analysed, so as to assert the biological plausibility of the searching behaviour?

There are many types of walking strategy that could be employed. Some examples are the Brownian walk (random orientated, uncorrelated runs), Correlated random walk (CRW) (correlation between runs indicating a tendency to continue movement in the same direction) and a Lévy walk (randomly orientated, uncorrelated runs with a different distribution of run lengths. Most runs are short, which are interrupted by occasional long runs) [3].

Nepomnyashchikh & Podgronyj claim that the biological plausibility of wandering behaviour can be measured by examining a group of non-interacting animals spreading from the same point. In their study, Mean Square Displacement (MSD) (9) is used as a method to quantitively analyse walks. Brownian walkers will have a Gaussian diffusion, while non-Brownian walks will have an MSD that increases with time resulting in 'anomalous' diffusion.

Before repeating Nepomnyashchikh & Podgronyj's dispersal comparison, the plausibility of searching behaviours becomes clear by examining the dispersal of five agents for each type. Figures 2 3 & 4 show the

typical patterns for Random, Sine and Logistic-map agents after 200 simulation steps. The square in the middle is the start point for the agents. The circle represents the end point.

The Sine agent is very jittery with little correlation in any of the motion. The agents' paths cross repeatedly and the agents constantly revisit areas previously searched. In contrast, both the Random agent (Figure 2) and the Logistic-map agent (Figure 4) show smoother paths and dispersal.

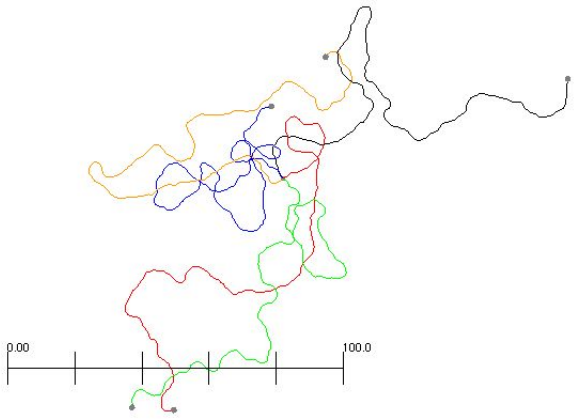

Figure 2: **Wandering paths for five random oscillator agents running over 200 simulation steps.**

4.1 Wandering Results

To analyse the diffusion of each agent type, 100 agents of each type are simulated, dispersing from a single point and their movements tracked over 10,000 simulation steps. The MSD for all the agents is calculated and displayed as a time series in Figures 5-7, where each figure contains the series for three different sets of agents and a line representing the Guassian/normal distribution (slope = 1).

4.2 Discussion

From the results we can see that both the Random agent and the Logistic-map agent both exhibit 'anamolous' diffusion while the Sine agent fairs poorly. The signals to the Sine agent's left and right motor can be seen to spike rapidly, switching between on and off quickly. This resulted in large turn angles at every step. No efficient searching behaviour is emergent from its dynamics.

Both the Random and Logistic-map agents' diffusion approaches Gaussian diffusion over time, so they both can be claimed to be biologically plausible, based on the criteria established in [13]. Interestingly, the Random agent displays CRW behaviour rather than the Brownian walking that would have been expected.

However, the nature of biologically plausibility based on the the comparison of diffusion to a random walking seems to be a poor methodology. Ryu and Samule showed that *Caenorhabditis elegans*, a type of roundworm, perform a Brownian walk initially before shifting to a Lévy walk after a period of about 15 minutes [15], when searching for food. For these worms, the diffusion is Gaussian at first and then anomalous, conflicting with Nepomnyashchikh & Podgronyj's test for biological plausibility.

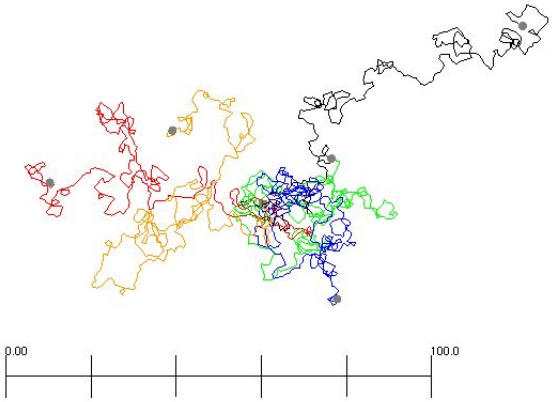

Figure 3: **Wandering paths for five sine oscillator based agents running over 200 simulation steps.**

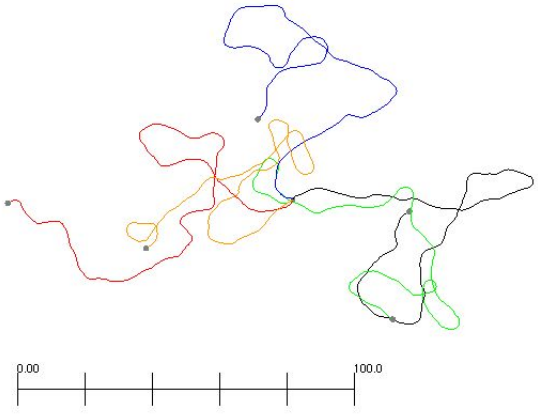

Figure 4: **Wandering paths for five logistic oscillator agents running over 200 simulation steps.**

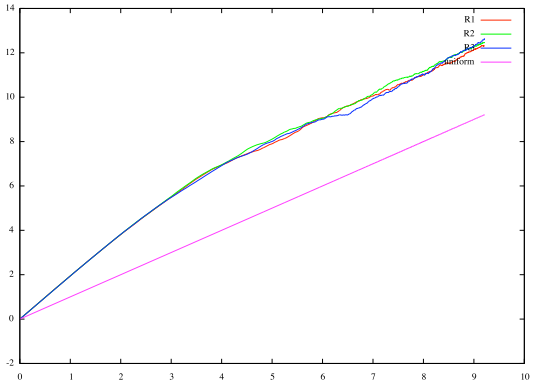

Figure 5: **Plot of log MSD vs. log time for 100 random oscillator agents.** 'uniform' represents the MSD of a Guassian distribution.

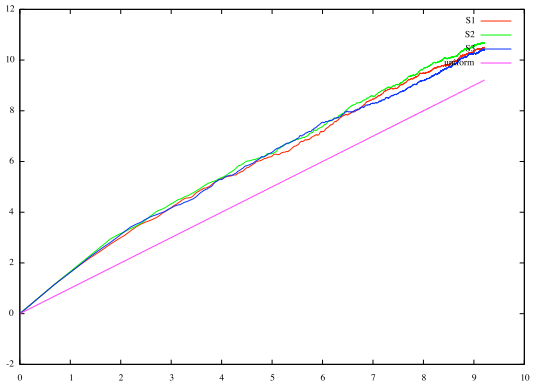

Figure 6: **Plot of log MSD vs. log time for 100 sine oscillator agents.** 'uniform' represents the MSD of a Guassian distribution.

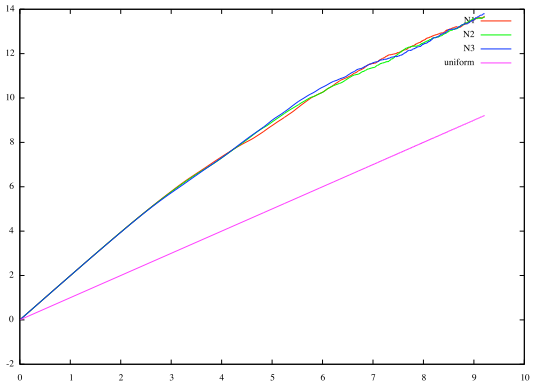

Figure 7: **Plot of log MSD vs. log time for 100 logistic oscillator agents.** 'uniform' represents the MSD of a Guassian distribution.

Orientation

In the second of Nepomnyashchikh & Podgronyj's tests, the agents are placed in a world with a 'Gaussian shaped odour field' at distances of 500, 1,000, 5,000 or 1,0000 away from the agent. The simulation is run until either the agent finds the centre of the odour field or has travelled for 100,000 steps. There is no comment about the size of the odour gradients used or the orientation of the agent. In [13] the maximum stimulus in the middle of the odour gradient is 100 and any unsuccessful run is replaced with an successful one. For each succesful run, the total distance travelled to the odour source is tracked.

When modelling the orientation task, the simulated world is given a single odour point with a Gaussian distribution in 2-dimensions, based on (11). In the initial runs the odour fields are around 400 units wide, with a standard deviation, $\sigma = 20$. Figure 8) shows the Gaussian distribution of values around the odour's centrepoint.

Results

The odour gradient was placed 500 units away from the starting points of the agents and each agent was given a start angle of 0 to maximise the chance of encountering the odour field. Figure 9 shows a typical path for a Logistic-map agent.

The agent initially encounters the odour field after a few hundred simulation steps, it then drifts across the odour and exits the field. In other runs agents reenter the field, drift across it again before, by luck, hitting the centre of the field and ending the simulation. The failure to perform chemotaxis in the presence of odour field stimulation can be explained by examining the design of the agents.

Each agent is equipped with a sensor that has inhibitory connections to the left and right turn oscillators. In addition each turn oscillator inhibits the other. The tendency for the left and right turn oscillators is to drive the other's output to zero. A high output in one inhibits a high output in the other and vice versa. Internal noise and the activation oscillator provide stimulus to temporarily excite the left and right turn oscillators.

As the agent encounters an odour gradient, the ambient stimulation begins to suppress the contributions of both turn oscillators (this can be seen in Figures 10-13). In addition, internal contributions from the activating oscillator and noise are swamped by the large input from the sensor. The agent is unable to turn when the odour gradient is strong and consequently it 'drifts' in a straight line across the odour gradient until the ambient stimulation drops to a point where excitatory input from noise or the activating oscillator allows the agent to break from this course and begin turning behaviour again.

This reflects Nepomnyashchikh & Podgronyj's statement that the observed 'movement... consists predominantly of straight runs that are rarely directed towards the source' and that the agent 'repeatedly leaves the source in various directions and returns to it again'. In their study the agents often circled or turned back into the field. Because the agent was observed 'sampling' the odour gradient they further use this as justification that the agent displays biologically plausible searching behaviour.

Vickers [17] shows that animals tend to employ locomotor tactics to perceive odour plumes, which are typically dynamic and dispersing in the real environment rather than a static simple geometric shape. So one could expect an agent to wander through an odour plume, exiting and entering the plume to gain information about it.

However, in the observed simulations the Logistic-map agents rarely reached the odour source (defined in [13] as the agent moving within a circle of 5 units radius from the centre of the field). Some would pass through the odour field multiple times, but not in a manner that indicated sampling or circling of the odour source.

Sine agents also performed in a similar manner. The failure of the Sine and Logistic-map agents to chemotaxis prevented the collection of orientation data.

Further work is required to identify why there is a discrepancy between the two sets of observed results.

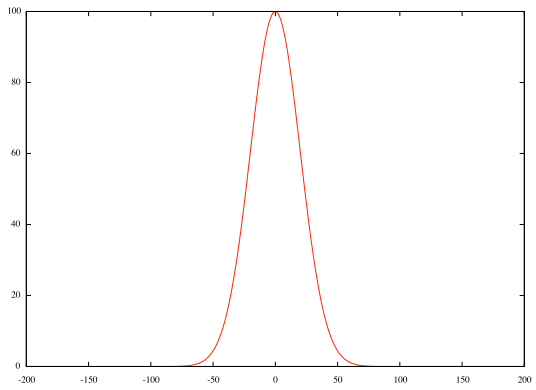

Figure 8: **Guassian distribution of values for odour point.** $\sigma = 20.0$

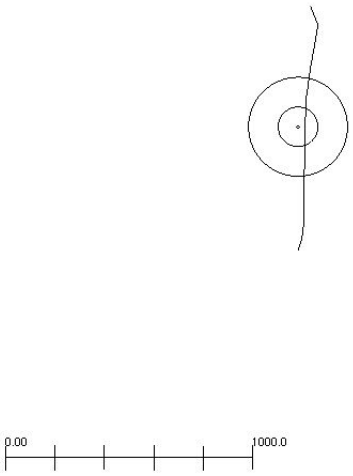

Figure 9: **Path of a logistic map agent encountering an odour gradient.** The inner circle represents the source of the food, the next circle the edge of the peak values for the gradients and the outer the edge of the field (where values have dropped below threshold).

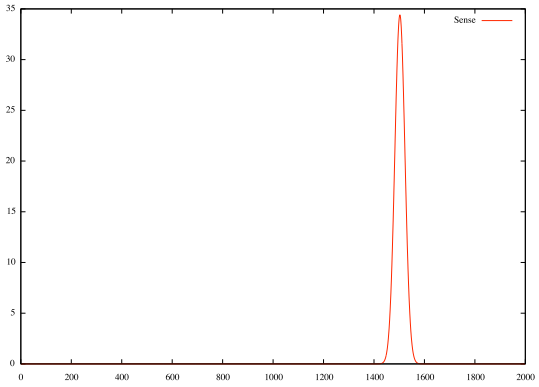

Figure 10: **Odour sensor signals.** The first 1000 steps occur before the agent is placed in the environment.

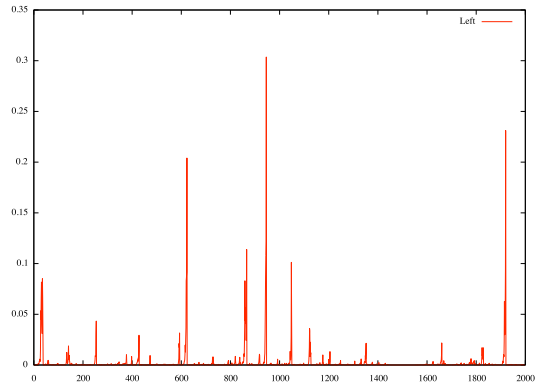

Figure 11: **Left turn oscillator signals.** The first 1000 steps occur before the agent is placed in the environment.

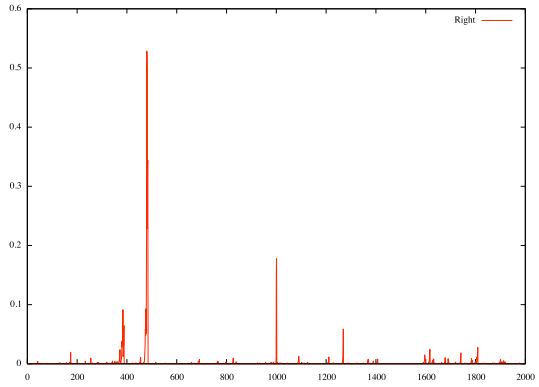

Figure 12: **Right turn oscillator signals.** The first 1000 steps occur before the agent is placed in the environment.

It is highly likely that the odour fields are just 'Gaussian shaped' rather than exhibiting strict Gaussian distribution, with a much different gradient and size than used in this study. The parameters of the agent need also to be checked to ensure that nothing is missing from the parameters described in [13].

Random agents, on the other hand, exhibit chemotaxis towards the food source, when they enter the edge of the odour field. Figure 14 shows an example path for one such agent. As the left and right turn oscillators are not connected by inhibitory connections, each oscillator acts independently, resulting in the staggering, random movement see in the wandering task. Any motion across the odour gradient steers the agent towards its centre. Movement away from the gradient excites turning, but turning into the gradient suppresses further turning, forcing the agent to head for the gradient's centre.

5 Conclusions

This study confirms the basis of Nepomnyashchikh & Podgronyj's work and clearly demonstrates that interesting searching behaviour can emerge from the physiology of the system. Basic behavioural traits can arise from the properties extant in the system based on the premise that all biological life has some form of CPG or other oscillatory components.

Without writing an explicit controller, is was shown that agents can exhibit biologically plausible wandering and orientation behaviours and it demonstrates a potentially rich area of exploitation for mobile robots. If CPG or other oscillator are present for walking control or for other reasons, then very basic behaviours may be available 'for free' based on the results of this study.

While repeating this study, a Random agent performed the wandering tasks in a potentially biologically plausible manner and showed chemotaxis in the orientation task. It was expected that a random uncorrelated strategy should exhibit a Brownian walk and therefore fail to explore the search space efficiently. The obvious question is why a pseudo-random agent failed to exhibit such behaviour. Although it's beyond the scope of this paper, the properties of the Mersenne Twister algorithm should be explored.

Problems with logistic maps.

Nepomnyashchikh & Podgronyj's agents are very simple but their dynamics are anything but. While Nepomnyashchikh & Podgronyj claim that their system is a 'simple non-linear system', the logistic map is a simple only in terms of its description. Its dynamics have an 'extraordinarily rich spectrum of dynamical behaviour, from stable points, through cascades of stable cycles to a regime in which the behaviour is... chaotic' [11].

The dynamics of coupled oscillators is a very broad field of research and has been studied for many years confirming that such systems have very rich and complex dynamics. Further research is therefore required to explore if there any advantages to using coupled logistic maps as oscillators over other circuits. If searching behaviours require oscillatory circuits, what other systems could be used and from a robot's point of view, what is the cost of implementation? The Sine-wave agent was an attempt to look at simpler oscillators but required a more rigorous implementation methodology to be of use.

Another problem with the logistic map approach of Nepomnyashchikh & Podgrony, is that time is not mentioned. The logistic map is sampled once per simulation step, but neither time nor frequency of the signal is mentioned.

An interesting side investigation would be look using a Continuous Time Neural Network (CTRNN) [4] to generate signals similar to the coupled logistic maps used by Nepomnyashchikh & Podgrony. Beer [5] dismisses logistic maps as saying that they are either '...dynamically universal, in which case their analysis will be in general as difficult as CTRNNs, or they are not, in which case they may miss important dynamical features'.

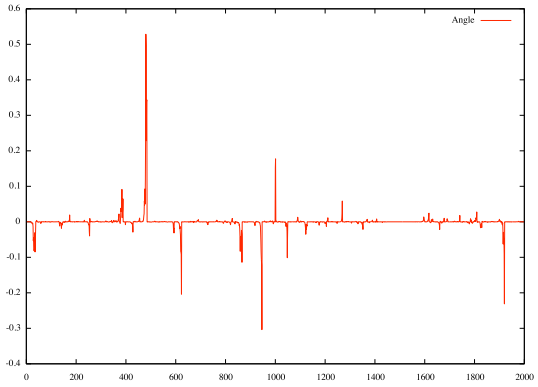

Figure 13: **Turn angle in radians.** The first 1000 steps occur before the agent is placed in the environment.

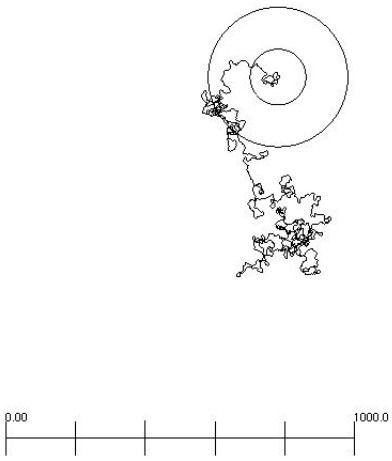

Figure 14: **Path of a random agent encountering an odour gradient.** As before the circles describe the odour gradient.

While CTRNNs are universal dynamical approximators that can be tuned to show the properties of any dynamical systems, it would be interesting to see what the smallest CTRNN is that shows semi-chaotic central pattern generation approximating the logistic map oscillators.

Explicit controllers

While no explicit control has been written for the Logistic-map agent, Nepomnyashchikh & Podgronyj have carefully tuned the parameters of their agent, enabling it to exhibit interesting behaviour. By predesignating the nodes and connections in their agents, they have carefully constrained the interactions so that phase transitions become manifest in their agents [13].

Further research should be performed to investigate what effect each parameter and the agent's morphology has on the tasks. Genetic algorithms should be employed to investigate to what extent the system has been designed, by comparing evolved solutions to the hand-tuned solution.

If we are indeed interested in the question of what is the simplest agent required for biologically plausible wandering behaviours then the evolutionary approach must be used. The Sine Agent was designed on a very limited search of its parameter space. As a consequence it suffered from motor control signals that switched rapidly between it maximum and minimum values resulting in the emergence of weak searching behaviour. An evolution approach should be used to evolve such agents so that the search space of all its parameters can be explored in a rigorous manner. Other simple signal types - cosine, saw, square, triangle waves should also be investigated by allowing morphological changes to evolve. An evolved Logistic-map agent may potentially also be simpler while solving the chemotaxis problems inherent in the current hand tuned solution.

Biological Plausibility

MSD is used by Nepomnyashchikh & Podgronyj as a basis to claim that the agents' diffusion is biologically plausible. However, biological agents can shift between different wandering behaviours and the choice of behaviour is dependent on the physiology of the agent, the medium through which its moving and interaction with con and hetrospecifics [2].

It has been suggested that for randomly distributed targets, it is more efficient to perform a Lévy walk than a Brownian walk. A Lévy walk is a random walk in which the run lengths have a power-law distribution (8), for large l, where $1 < \mu < 3$. A Brownian walk is a random walk in which the run lengths have an normal distribution ($\mu \geq 3$) [2]. More specifically it has been shown that for sparsely distributed targets, the optimal value of μ is 2.

$$P(l) = l^{-\mu} \tag{8}$$

Therefore a potentially more useful focus would be to examine emergence of efficient searching behaviours, including switching strategies dependent on time or median, from the dynamics of the internal system and the quantitive analysis of those behaviours [2] [18].

References

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

[2] Bartumeus, F., da Luz, M.G.E., Viswanathan, M. & Catalana, J. (2005) *Animal Search Strategies: A quantitative random-walk analysis.* Ecology, Vol. 86(11):3078-3087.

[3] Bartumeus, F. (2005) *Lévy Processes in Animal Movement and Dispersal.* Ph.D. Thesis, University of Barcelona.

[4] Beer, R.D. (1996), *Towards the evolution of dynamical neural networks for minimally cognitive behavior.* Proc. 4th Int. Conf. on Simulation of Adaptive Behavior: 421-429. MIT Press.

[5] Beer, R.D. (2003) *Arches and stones in cognitive architecture.* Adaptive Behavior, Vol 11(4): 299-305.

[6] Braitenberg, V. (1984) *Vehicles: Experiments in synthetic psychology.* MIT Press.

[7] Chiel, H. J., Beer, R. D., and Gallagher, J. G. (1999). *Evolution and analysis of model CPGs for walking: Dynamical modules.* Journal of Computational Neuroscience, Vol 7(2): 99118.

[8] Freeman, W.J. (1987). *Simulation of chaotic EEG patterns with a dynamic model of the olfactory system.* Biological Cybernetics, Vol 56(2): 139-150.

[9] Takashi Ikegami (2006) *From Homeostasis to Sensory motor Couplings.* Presentation in Behaviour and Mind as a Complex Adaptive Systems workshop, SAB 2006.

[10] Independent JPEG Group, http://www.ijg.org/.

[11] May, R.M. (1976) *Simple mathematical models with very complicated dynamics.* Nature, Vol. 261: 459-467.

[12] Mobus, G.E. & Fisher, P.S. (1999) *Foraging search at the edge of chaos.* In Oscillations in Neural Networks, Lawrence Erlbaum Associates.

[13] Nepomnyashchikh, V.A. & Podgronyj, K.A. (2003), *Emergence of adaptive searching rules from the dynamics of a simple non-linear system.* Adaptive Behavior, Vol 11(4): 245-265.

[14] Press, W.H, Flannery, B.P., Teukolsky, S.A. & Vetterling, W.T. (2002) *Numerical recipes in C: The art of scientific computing.* 2nd Ed, Cambridge University Press.

[15] Ryu, W.S. & Samuel, A.D.T. (2002) *Thermotaxis in Caenorhabditis elegans analysed by measuring response to thermal stimuli.* Journal of Neuroscience, Vol. 22, 13:5727-5733.

[16] Varela, F. J. (1992), *Autopoesis and the biology of intentionality.*, In B. McMullin (ed), Proceedings of the Workshop Autopoiesis and Perception, Dublin City University: 4-14.

[17] Vickers, N.J. (2000) *Mechanisms of animal navigation in odor plumes.* Biology Bulletin, 198: 203212.

[18] Viswanathan, G., Buldyrev, S.V., Havlin, S., da Luz, M.G.E., Raposo, E.P. & Stanley, H.E. (1999) *Optimizing the success of random searches.* Nature, Vol 401:911-913.

[19] Wagner, R. (2003) *Mersenne Twister C++ Class.* http://www-personal.engin.umich.edu/ wagnerr/MersenneTwister.html.

Appendix - Equations

Mean Squared Displacement

The mean squared displacement for a two dimensional system is calculated by (9), where (x_0, y_0) is the starting reference point, N is the number of agents, t is a given time, $(x_i(t), y_i(t))$ is the point of agent i at time t.

$$\langle R(t)^2 \rangle = \frac{1}{N} \sum_{i=1}^{N} [(x_i(t) - x_0)^2 + (y_i(t) - y_0)^2] \tag{9}$$

$$\langle R(t)^2 \rangle \approx t^{\alpha} \tag{10}$$

The scaling behaviour of MSD with time is given by (10) and characterises the spreading rate of the diffusion process. A value of $\alpha = 1$ indicates normal diffusion, $\alpha \neq 0.$ indicates anomalous diffusion [3].

Gaussian Distribution

Using an isotropic (i.e. circularly symmetric) Gaussian distribution of the form (11), where σ is the standard deviation of the distribution.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y2}{2\pi\sigma^2}} \tag{11}$$

Appendix B - Code

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

// forward ref
class World;

/**
 * @class Agent
 * An abstract class representing the basic interface of agents in
 * the simulation
 */
class Agent
{
public:
        Agent();
        virtual ~Agent();

        /**
         * Initialise the agent at a given point and orientation
         * @param       start   The starting location of the agent
         * @param       angle   The startint direction for the agent
         * @return True if agent is initialised, false otherwise
         */
        virtual bool            Init            (const Point& start, float angle) = 0;

        /**
         * Embedded the agent in a world
         * @note World pointer is not ownder by Agent
         * @param world  The world to embedded the agent in
         * @return True if agent is embedded, false otherwise
         */
        virtual bool            Embed           (World* world);

        /**
         * Run 'one unit' of time and return agents new position
         * @return True if agent is ok, false to stop the simulation
         */
        virtual bool            RunOneUnit      () = 0;

        /**
         * Get the current position of the agent
         * @return The position of the agent
         */
        Point   Pos             (void) const;

        /**
         * Get the current orientation of the agent
         * @return The angle (between 0-360 deg) the agent is facing
         */
        float   Angle   (void) const;

        /**
         * Request information about the world at the current point
         * @param info  Information about that position
         * @return      True if the world was sampled correctly
         */
        bool    Sense   (WorldInfo& info);

protected:
```

```
        /**
         * Set the positon of the agent
         * @param pos   The new position of the agent
         */
        void    SetPos  (const Point& pos);

        /**
         * Set the direction of the agent
         * @param angle The new angle of the agent
         */
        void    SetAngle(float angle);

        /**
         * Shared code for calculating new position based on turn angle
         * in radians and the number of units to travel
         * @param turnAngleRad  The ammount to turn, in radians
         * @param distance              The distance to travel
         * @return True if updated, false otherwise
         */
        bool UpdatePosAngle (float turnAngleRad, float distance);

private:
        Point   m_pos;                  //!< Position of the agent
        float   m_angle;                //!< Direction of the agent
        World*  m_world;                //!< World containing the agent [Not owned by object]
};
```

```
/*
 *  Agent.cpp
 */

#include <math.h>
#include "Globals.h"
#include "Geo.h"
#include "Utils.h"
#include "World.h"
#include "Agent.h"


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

Agent::Agent() : m_angle(0.0), m_world(NULL)
{
        m_pos.x = 0.0;
        m_pos.y = 0.0;
}

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

Agent::~Agent()
{
}

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

bool Agent::Init (const Point& start, float angle)
{
        m_pos.x = start.x;
        m_pos.y = start.y;
        m_angle = NormaliseAngle(angle);

        return true;
}
```

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

Point Agent::Pos (void) const {
        return m_pos;
}

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

void Agent::SetPos (const Point& pos) {
        m_pos = pos;
}

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

float Agent::Angle      (void) const {
        return m_angle;
}

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

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

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

bool Agent::Embed(World* world) {
        m_world = world;
        return true;
}

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

bool Agent::UpdatePosAngle (float turnAngleRad, float distance)
{
        float turnAngleDeg  = RadiansToDegrees(turnAngleRad);
        float newAngle          = NormaliseAngle(Agent::Angle() + turnAngleDeg);

        // Calc new position
        // Move distance based on new heading
        // C sin and cos take angle in rad so convert back
        float x_delta = (sin(newAngle * kPI/180.0)) * distance;
        float y_delta = (cos(newAngle * kPI/180.0)) * distance;

        // Calculate new point
        Point pos = Agent::Pos();

        pos.x += x_delta;
        pos.y += y_delta;

        Agent::SetPos(pos);
        Agent::SetAngle(newAngle);

        return true;
}

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

bool Agent::Sense (WorldInfo& info)
{
        bool ok = false;

        if (m_world) {
                ok = m_world->Sense(m_pos, info);
```

```
        }
        return ok;
}
```

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

#include "Node.h";

/**
 * @class Connection
 * Represents a one-way connection between two nodes
 */
class Connection
{
public:
        /**
         * Constructor
         * @param from  The starting point of the connection
         * @param to    The end point of the connection
         * @param ctype The type of the connection
         */
        Connection(Node* from, Node* to, Node::ConnectionType ctype);

        /**
         * Destructor
         * Do nothing, nodes are not owned by connection
         */
        ~Connection() {}

        /**
         * Reads from the connection
         * @return Returns the value of the connection
         */
        float Read () const;

        /**
         * Sets the weight of the connection
         * @param weight        The new connection weight
         */
        void  SetWeight (float weight);

        /**
         * Gets the weight of the connection
         * @return      The weight of the connection
         */
        float GetWeight (void)  const;

private:
        Node*                                   m_from;         //!< Start point of the connection
        Node*                                   m_to;           //!< End point of the connection
        Node::ConnectionType    m_type;         //!< Type of connection
        float                                   m_weight;       //!< Weight of connection
};
```

```
/*
 *  Connection.cpp
 */
```

```
#include "Node.h"
#include "Connection.h"

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

Connection::Connection(Node* from, Node* to, Node::ConnectionType cType) :
m_from(from), m_to(to), m_type(cType), m_weight(1.0)
{
}

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

float Connection::Read () const
{
        float val = 0.0;
        if (m_from) {
                val = m_from->NodeValue();

                // If connection is inbitory then value must
                // be subtracted
                if (m_type == Node::Inhibitory)
                        val = -val;
        }
        // Finally, multiply by connection weight
        val *= m_weight;
        return val;
}
```

```
/*
 *  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
 */
#include <stdlib.h>
#include <stdio.h>
#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;
}
```

```
/*
 *  Geo.h
 */

#pragma once

#include <vector>

/**
 * Simple 2d point structure
 */
typedef struct
{
        float x;
        float y;
}
Point;

/**
 * Simple 2d rectangle structure
 */
typedef struct
{
        Point bottom;           //!< Bottom left point
        Point top;                      //!< Top right point
}
Rect;

typedef enum
{
        kLINE_TO,                       // path from current point to p1
        kRECT_TO,                       // rect with p1 (bottom) or p2 (top)
        kCURVE_TO,                      // pdf curveto operator
        kMOVE_TO,                       // Close current path and move point to p1
        kPOINT_TO,                      // Moves current point, moves to point p1 and plots point
        kCIRCLE_PATH,           // Draws a line circle at p1, using p2.x as radius
        kEND_PATH,                      // End path, without closing it
        kCLOSE_PATH                     // End path, but close path to start point
}
PathType;

typedef struct
{
        Point   p1;
        Point   p2;
        Point   p3;
        PathType type;
}
Path;

/**
 * An array of Point objects
 */
typedef std::vector<Point>      Points;
```

```
/**
 * An array of Path objects
 */
typedef std::vector<Path>       Paths;

/**
 * Calculates the mean square displacement of an array of points based on a start point
 * @param points        An array of points
 * @param start         The start point to compare agains
 * @return The MSD for the points
 */
float MeanSquareDisplacement (const Points& points, Point start);

/**
 * Gaussian Distribution object
 * Calculates the value at a given point in space away from the anchor
 * based on a Gaussian Distribution.
 */
class GaussianDistribution
{
public:
        /**
         * Default constructor
         * Sets up distribution with standard deviation = 1.0, no amplication
         * no threshold and anchored at {0.0, 0.0}
         */
        GaussianDistribution ();

        /**
         * Constructor
         * @param stdDev        The standard deviation of the gaussian distribution
         * @param amp           A value amplication factor for gaussian values
         * @param threshold     A cutoff value, below this the output is 0.0
         * @param anchor        The anchor point of the distribution
         */
        GaussianDistribution (float stdDev, float amp, float threshold, Point anchor);

        /**
         * Calculate the value at this point based on a gaussian distribution
         */
        float   Val (Point pos);

private:
        Point m_anchor;         //!< Point the distribution is anchored at
        float m_threshold;      //!< Threshold for values, below this the value becomes 0.0
        float m_stdDev;         //!< Standard deviation of the distribution
        float m_amp;            //!< Amplifying constant to apply to all values in the distribution
};

/**
 * Investigate nature of Gaussian Distribution
 * @param width         Will examine guassian from -width,width range.
 * @param stdDev        The standard deviation of the gaussian distribution
 * @param amp           A value amplication factor for gaussian values
 * @param threshold     A cutoff value, below this the output is 0.0
 * @return The width of the distrubtion
 */
int GaussianTest (int width, float stdDev, float amp, float threshold);
```

```
/*
 *  Geo.cpp
 */
```

```
#include <iostream>
#include <fstream>

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

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

float MeanSquareDisplacement (const Points& displacements, Point start)
{
        float msd = 0.0;
        float sum = 0.0;
        int n = 0;
        Points::const_iterator pos = displacements.begin();
        for ( ; pos != displacements.end(); ++pos) {
                Point point = (*pos);

                // (x(t) - x(0))^2 + (y(t) - y(0))^2
                sum +=  pow((point.x - start.x), 2.0) + pow((point.y - start.y), 2.0);
                n++;
        }
        msd = sum / n;
        return msd;
}

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

GaussianDistribution::GaussianDistribution() : m_threshold(0.0),
m_stdDev(1.0), m_amp(1.0)
{
        m_anchor.x = 0.0;
        m_anchor.y = 0.0;
}

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

GaussianDistribution::GaussianDistribution
(float stdDev, float amp, float threshold, Point anchor) :
m_anchor(anchor), m_threshold(threshold), m_stdDev(stdDev), m_amp(amp)
{
}

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

float GaussianDistribution::Val (Point pos)
{
        // shift to origin (ie translate away from anchor point)
        pos.x -= m_anchor.x;
        pos.y -= m_anchor.y;

        float twoSig = (2 * pow(m_stdDev, 2.0));
        float part1 = 1.0 / (kPI * twoSig);
        float part2 = (pow(pos.x, 2.0) + pow(pos.y, 2.0)) / twoSig;
        float val   = part1 * exp(-part2);

        // amplify val
        val *= m_amp;

        // if val is less than our threshold return nothing
        val = (val < m_threshold ? 0.0 : val);

        return val;
}
```

```
//-------------------------------------------------------
// Used to test width and values of a gaussian distribution
int GaussianTest (int guessedWidth, float stdDev, float amp, float threshold)
{
        // Set up odour gradient
        Point pos = {0.0, 0.0};

        // StdDev of 5.0, means max value is = 0.006366
        // to get 100, 100/0.006366 = 15,708.4511467169
        // Width is approx 141 units wide
        GaussianDistribution* gd = new GaussianDistribution(stdDev, amp, threshold, pos);

        float max = 0.0;
        int width = 0;
        float val;

        std::ofstream gStream;
        gStream.open("/Users/pip/Desktop/Gaussian.Dat");

        gStream << "#plot \"Gaussian.dat\"";
        gStream << " using 1:2 title 'Gauss' with lines" << std::endl;

        for (int i = -guessedWidth; i < guessedWidth; i++) {
                Point testPos = {i, 0.0};
                val =  gd->Val(testPos);

                gStream << i << "\t" << val << std::endl;

                if (val > 0.0) {
                        width++;
                }

                if (val > max) {
                        max = val;
                }
        }
        gStream.close();

        delete gd;
        return width;
}
```

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

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

/**
 * Number of steps to run the oscillators before starting simulation
 */
#define kSIMULATION_DELAY    1000

/**
 * Number of steps to run for simulation. Defined here so it can be
 * shared with static methods
 */
#define kSIMULATION_STEPS       1000

#define kWindowWidth            640             //!< Width of the OpenGL window
#define kWindowHeight           480             //!< Height of the OpenGL window
```

```
/**
 * Default path for file export
 */
#define kDEFAULT_FILE_PATH      "/Users/pip/Desktop/"

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

/**
 * N&P state their agent moves 1 unit in each turn
 */
#define kDISTANCE_MOVED 1.0
```

```
/*
 *  NPAgent.h
 */

#pragma once

// Forward Ref
class Random;
class Sensor;

class NPAgent : public Agent
{
public:
        /**
         * Default Constructor
         * Uses default values for mu and peak == 1.1
         * as defined in N&P paper
         */
        NPAgent ();

        /**
         * Constructor
         * @param mu    The average parameter
         * @param peak  The peak magnitude parameter of internal noise
         */
        NPAgent (float mu, float peak);

        ~NPAgent();

        //---------------------------------------
        // Agent interface
        //---------------------------------------
        bool    Init            (const Point& start, float angle);
        bool    RunOneUnit      ();

private:
        Node*   m_left;                 //!< The left turn oscillator
        Node*   m_right;                //!< The right turn oscillator
        Sensor* m_sensor;               //!< Magic odour sensor
        Node*   m_activation;   //!< The activation oscillator

        float   m_mu;                   //!< The average parameter
        float   m_peak;                 //!< Peak magnitude of internal noise
};
```

```
/*
```

```
 *  NPAgent.cpp
 */

/**
 * If true log agent values
 */
#define AGENT_LOGGING 1

#if AGENT_LOGGING
#include <iostream>
#include <fstream>
#endif

#include <math.h>

#include "Globals.h"
#include "Geo.h"
#include "Node.h"
#include "World.h"
#include "Agent.h"
#include "NPAgent.h"
#include "NPOscillator.h"

#include "Random.h"
#include "RandomNormal.h"
#include "Utils.h"
#include "Sensor.h"

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

/**
 * N&P state that for much of their paper they keep mu and peak at 1.1
 */
#define kDEFAULT_MU 1.1

/**
 * N&P state that for much of their paper they keep mu and peak at 1.1
 */
#define kDEFAULT_SIGMA 1.1


/*
 * N&P define a the min value for an oscillator to be 10^-6
 */
#define kOSCILLATOR_MIN_VALUE 0.0001

/*
 * N&P define a the max value for an oscillator to be 1.0
 */
#define kOSCILLATOR_MAX_VALUE 1.0


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

#if AGENT_LOGGING
std::ofstream leftstream;
std::ofstream rightstream;
std::ofstream activationstream;
std::ofstream turnStream;
std::ofstream sensStream;
static int stepcount = 0;

static void OpenLogs (void);
static void CloseLogs (void);
#endif  // kAGENT_LOGGING
```

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

NPAgent::NPAgent() : m_left(NULL), m_right(NULL), m_sensor(NULL), m_activation(NULL),
m_mu(kDEFAULT_MU), m_peak(kDEFAULT_SIGMA)
{
#if AGENT_LOGGING
        OpenLogs();
#endif // AGENT_LOGGING
}

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

NPAgent::NPAgent(float mu, float peak) :
m_left(NULL), m_right(NULL), m_sensor(NULL), m_activation(NULL),
m_mu(mu), m_peak(peak)
{
#if AGENT_LOGGING
        OpenLogs();
#endif // AGENT_LOGGING
}

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

NPAgent::~NPAgent()
{
#if AGENT_LOGGING
        CloseLogs();
#endif // AGENT_LOGGING
        delete m_sensor;
        delete m_right;
        delete m_left;
        delete m_activation;
}

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

bool NPAgent::Init (const Point& pos, float angle)
{
        Random rand;
        bool ok = true;

        Agent::Init(pos, angle);

        // Create Left turn oscillator
        RandomNormal* l_noise = new RandomNormal();
        NPOscillator* l_motor = new NPOscillator(m_mu, m_peak, l_noise);
        NormaliseMaxMin<float>* l_funct = new NormaliseMaxMin<float>
                                                                        (kOSCILLATOR_MAX_VALUE, kOSCILLATOR_MIN_VALUE);
        l_motor->Normalize(l_funct);
        l_motor->SetNodeValue(rand.Rand());             // set l(0)
        m_left                  = l_motor;

        // Create right turn oscillator
        RandomNormal* r_noise = new RandomNormal();
        NPOscillator* r_motor = new NPOscillator(m_mu, m_peak, r_noise);
        NormaliseMaxMin<float>* r_funct = new NormaliseMaxMin<float>
                                                                        (kOSCILLATOR_MAX_VALUE, kOSCILLATOR_MIN_VALUE);
        r_motor->Normalize(r_funct);
        r_motor->SetNodeValue(rand.Rand());             // Set r(0)
        m_right                 = r_motor;

        // Create activation oscillator
        RandomNormal* a_noise = new RandomNormal();
        NPOscillator* a_node  = new NPOscillator(m_mu, m_peak, a_noise);
```

```
        NormaliseMaxMin<float>* a_funct = new NormaliseMaxMin<float>
                                                                        (kOSCILLATOR_MAX_VALUE, kOSCILLATOR_MIN_VALUE);
        a_node->Normalize(a_funct);
        a_node->SetNodeValue(0.01);                             // set a(0)
        m_activation             = a_node;

        // Create new sensor
        m_sensor = new Sensor();
        m_sensor->Attach(this);                 // attach the agent to the senor

        // Create the connections between them
        m_left->ConnectTo(m_right, Node::Inhibitory);
        m_right->ConnectTo(m_left, Node::Inhibitory);

        m_activation->ConnectTo(m_right, Node::Excitatory);
        m_activation->ConnectTo(m_left,  Node::Excitatory);

        m_sensor->ConnectTo(m_right, Node::Inhibitory);
        m_sensor->ConnectTo(m_left,  Node::Inhibitory);

        // Randomise x(0) values for oscillators
        return ok;
}

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

bool NPAgent::RunOneUnit ()
{
        // Generate new values
        float newAct = m_activation->CalcNodeValue();
        float newL       = m_left->CalcNodeValue();
        float newR       = m_right->CalcNodeValue();
        float newS       = m_sensor->CalcNodeValue();

        // Calculate Turn position
        // N&P use the difference of the left and right values as
        // the turn angle (in radians)
        float tRad   = newR - newL;
        Agent::UpdatePosAngle(tRad, kDISTANCE_MOVED);

        // Update nodes
        m_activation->SetNodeValue(newAct);
        m_left->SetNodeValue(newL);
        m_right->SetNodeValue(newR);
        m_sensor->SetNodeValue(newS);

#if AGENT_LOGGING
        leftstream        << stepcount << "\t" << newL   << std::endl;
        rightstream       << stepcount << "\t" << newR   << std::endl;
        activationstream  << stepcount << "\t" << newAct << std::endl;
        turnStream        << stepcount << "\t" << tRad   << std::endl;
        sensStream        << stepcount << "\t" << newS   << std::endl;
        stepcount++;
#endif
        return true;
}




//---------------------------------------------------------------------
// Log utility code
//---------------------------------------------------------------------
#if AGENT_LOGGING
```

```
void OpenLogs ()
{
        // reset counter
        stepcount = 0;

        // Open left oscillator log
        std::string buffer = kDEFAULT_FILE_PATH;
        buffer += "LeftTurn.dat";
        leftstream.open(buffer.c_str());
        leftstream << "#plot \"LeftTurn.data\" using 1:2 title 'Left' with lines" << std::endl;
        leftstream << "#Step\tLeft" << std::endl;

        // Open right oscillator log
        buffer = kDEFAULT_FILE_PATH;
        buffer += "RightTurn.dat";
        rightstream.open(buffer.c_str());
        rightstream << "#plot \"RightTurn.data\" using 1:2 title 'Right' with lines" << std::endl;
        rightstream << "#Step\tRight" << std::endl;

        // Open activating oscillator log
        buffer = kDEFAULT_FILE_PATH;
        buffer += "Activation.dat";
        activationstream.open(buffer.c_str());
        activationstream << "#plot \"Activation.data\" using 1:2 title 'Activation' with lines";
        activationstream << std::endl << "#Step\tActivation" << std::endl;

        // Open turn angle log
        buffer = kDEFAULT_FILE_PATH;
        buffer += "TurnAngle.dat";
        turnStream.open(buffer.c_str());
        turnStream << "#plot \"TurnAngle.data\" using 1:2 title 'Angle' with lines" << std::endl;
        turnStream << "#Step\tAngle" << std::endl;

        // Open sensor log
        buffer = kDEFAULT_FILE_PATH;
        buffer += "SenseData.dat";
        sensStream.open(buffer.c_str());
        sensStream << "#plot \"SenseData.dat\" using 1:2 title 'Sense' with lines" << std::endl;
        sensStream << "#Step\tSense" << std::endl;
}

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

void CloseLogs ()
{
        leftstream.close();
        rightstream.close();
        activationstream.close();
        turnStream.close();
        sensStream.close();
}

#endif
```

```
/*
 *  LogisticNode.h
 */

#pragma once

// Forward ref
class Random;
#include "Utils.h"
```

```
/**
 * @class NPOscillator
 * A logistic map based oscillator as defined
 * by N&P's paper
 */
class NPOscillator : public Node
{
public:
        NPOscillator(float m_mu, float m_peak, Random* noise);
        ~NPOscillator();

        //------------------------------
        // Override Node methods
        float   CalcNodeValue   () const;
        void    SetNodeValue    (float val);

        void    Normalize (Transform<float>* funct);
        float   Lambda  () const;

private:
        float           m_mu;   ///<! Value of
        float           m_peak;
        Random*         m_noise;
        Transform<float>*
                                m_normal;
};
```

```
/*
 *  LogisticNode.cpp
 */

#include <cmath>

#include "Random.h"
#include "Node.h"
#include "NPOscillator.h"
#include "Utils.h"

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

NPOscillator::NPOscillator(float mu, float peak, Random* noise) :
Node("Logistic"), m_mu(mu), m_peak(peak), m_noise(noise), m_normal(NULL)
{
}

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

NPOscillator::~NPOscillator()
{
        delete m_noise;
        delete m_normal;
}

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

void NPOscillator::SetNodeValue (float val)
{
        // Call any normalising object attached to node
        // Used to keep values in range
        if (m_normal) {
                val = m_normal->Op(val);
        }
```

```
        Node::SetNodeValue(val);
}

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

void NPOscillator::Normalize (Transform<float>* funct)
{
        delete m_normal;
        m_normal = funct;
}

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

float NPOscillator::CalcNodeValue () const
{
        float x_i = Node::NodeValue();
        float val = Lambda() * x_i;
        val *= (1 - x_i);
        return val;
}

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

float NPOscillator::Lambda () const
{
        float lam = m_mu;

        if (m_noise) {
                float r = m_noise->Rand();
                float noise = r * m_peak;
                lam += noise;
        }

        if (IsNan(lam)) {
                assert(0);
        }

        // Read from connections
        lam += Node::InputValue();

        // Finally check that lambda is in range
        // as defined by N&P paper
        if (lam < 0.0)
                lam = 0.0;
        else if (lam > 4.0)
                lam = 4.0;
        return lam;
}
```

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

#include <string>
#include <vector>

// forward ref
class Connection;

/**
```

```
 * @class Node
 * A basic subunit of an agent
 */
class Node
{
        typedef std::vector<Connection*> Connections;

public:

        typedef enum
        {
                Inhibitory,
                Excitatory
        }
        ConnectionType;

        Node (const char *name);
        virtual ~Node();

        /**
         * Returns the value for this node
         * @return The current node value
         */
        float                   NodeValue               () const;

        /**
         * Sets the value for this node
         * @param val   The new value for this node
         */
        virtual void    SetNodeValue    (float val);

        /**
         * Calculates the value for the node at next time step.
         * @return The node at the next time step
         */
        virtual float   CalcNodeValue   () const = 0;

        /**
         * Returns the sum of all the inputs to the node
         * Excitatory connections result in postive input, inhibitory in
         * negative. Values can also be weighted.
         * @return Sum of all the inputs.
         */
        virtual float   InputValue      () const;

        /**
         * Add a connection from this node to a new node
         * @param toNode        The node to connect to
         * @param cType         The connection type: excitatory or inhibitory
         * @return True if a successful connection was made, false otherwise
         */
        virtual bool    ConnectTo       (Node* toNode, ConnectionType cType);

        /**
         * Add a connection from a node to this node
         * @param fromNode      The node to connect from
         * @param cType         The connection type: excitatory or inhibitory
         * @return True if a successful connection was made, false otherwise
         */
        virtual bool    ConnectFrom     (Node* fromNode, ConnectionType cType);


private:
        float           m_value;                        ///!< Current value of the node
        std::string m_name;                             ///!< Nodes name for display/id
```

```
        Connections     m_connections;          ///!< Input connections to this node.
};
```

```
/*
 *  Node.cpp
 */

#include "Node.h"
#include "Connection.h"

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

Node::Node (const char *name) : m_name(name), m_value(0.0)
{
}

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

Node::~Node()
{
        while (!m_connections.empty()) {
                Connection* ptr = m_connections.back();
                m_connections.pop_back();
                delete ptr;
        }
}

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

float Node::NodeValue () const
{
        return m_value;
}

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

void Node::SetNodeValue (float val)
{
        m_value = val;
}

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

float Node::CalcNodeValue () const
{
        return m_value;
}

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

float Node::InputValue () const
{
        float val = 0.0;

        Connections::const_iterator pos = m_connections.begin();
        for ( ; pos != m_connections.end(); ++pos) {
                val += (*pos)->Read();
        }
        return val;
}

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

```
bool Node::ConnectTo (Node* toNode, Node::ConnectionType cType)
{
        if (toNode) {
                // Connection object owned by to node (node at tail of connection)
                Connection* c_ptr = new Connection(this, toNode, cType);
                toNode->m_connections.push_back(c_ptr);
        }
        return true;
}

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

bool Node::ConnectFrom (Node* fromNode, ConnectionType cType)
{
        if (fromNode) {
                fromNode->ConnectTo(this, cType);
        }
        return true;
}

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

```
/*
 *  OdourWorld.h
 */

 #pragma once

 // forward ref
 class GaussianDistribution;

 /**
  * Minimal world to run agents within
  * Has odour gradients
  *
  */
 class OdourWorld : public World
 {
 public:
        OdourWorld(Rect& rect);
        ~OdourWorld();

        //----- World Interface ------------------
        bool    Init    (void);
        bool    Run             (int num, int s);
        bool    Sense   (Point pos, WorldInfo& info);

private:
        GaussianDistribution*   m_odour;                //!< Source of odour in the world
 };
```

```
/*
 *  OdourWorld.cpp
 */

#include "Globals.h"
#include "Geo.h"
#include "World.h"
#include "OdourWorld.h"
```

```
/*
 *  RandomWorld.cpp
 *  ALife
 *
 *  Created by pip on 07/01/2007.
 *  Copyright 2007 __MyCompanyName__. All rights reserved.
 *
 */

#if DEBUG
#include <iostream>
#include <fstream>
#endif

#include <sstream>
#include <string>
#include <vector>

#include "Globals.h"
#include "math.h"
#include "Geo.h"
#include "Node.h"
#include "World.h"
#include "Agent.h"
#include "Random.h"
#include "RandomAgent.h"
#include "SimpleAgentWorld.h"
#include "Utils.h"

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

OdourWorld::OdourWorld(Rect& rect) : World(rect), m_odour(NULL)
{
}

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

OdourWorld::~OdourWorld()
{
        delete m_odour;
}

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

bool OdourWorld::Init ()
{
        return true;
}

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

bool OdourWorld::Sense  (Point pos, WorldInfo& info)
{
        if (m_odour) {
                info.data = m_odour->Val(pos);
        }
        return true;
}

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

bool PointNear (Point p1, Point p2, float distance)
{
        bool ok = false;
        if ( (p1.x > (p2.x - distance)) && (p1.x < (p2.x + distance)) ) {
```

```
                if ( (p1.y > (p2.y - distance)) && (p1.y < (p2.y + distance)) ) {
                        ok = true;
                }
        }
        return ok;
}

//-------------------------------------------------------
// Logs full path of agents
bool OdourWorld::Run (int num_agents, int steps)
{
        bool  ok       = true;

        // Set up odour gradient, 500 units north of where agent starts
        Point gPos = {0.0, 500.0};

        // StdDev of 20.0, means max value is approx = 0.000397887372
        // to get 100, 100/0.000397887372 = 251,327.403273306
        // Width is approx 400 units wide
        const float width = 400.0;
        const float centre_width = 5.0;

        m_odour = new GaussianDistribution(20.0, 251327.4, 0.0, gPos);

        // Add to drawing list
        Path path;

        // Add width of odour
        memset(&path, 0, sizeof(path));
        path.p1   = gPos;
        path.p2.x = width / 2.0;        // radius
        path.type = kCIRCLE_PATH;
        World::AddPath(path);

        // Add centre
        memset(&path, 0, sizeof(path));
        path.p1   = gPos;
        path.p2.x = centre_width;
        path.type = kCIRCLE_PATH;
        World::AddPath(path);

        // Add range of large values
        memset(&path, 0, sizeof(path));
        path.p1   = gPos;
        path.p2.x = (20.0 * 4.0);
        path.type = kCIRCLE_PATH;
        World::AddPath(path);

        Random rand;                                    // Grab a random number generator
        float distance = 0.0;                   // Distance travelled by agent
        int   successfull_agents = 0;   // Number of succesful agents
        Paths agent_paths;                              // Agent's path in the enivronment
        agent_paths.reserve(steps);

        while (successfull_agents < num_agents) {
                // track success
                bool found = false;

                // Clear previous agents data
                distance = 0.0;

                // Start agent in middle
                float x         = 0.0;
                float y         = 0.0;
```

```
Point pos   = {x, y};
// float angle = rand.Rand() * 360.0;   // generate random angle
float angle = 0.0;                                          // Head towards food source

// Add new agent
Agent * agent = World::GetAgent();

if (agent == NULL)
        break;

agent->Init(pos, angle);
agent->Embed(this);

// Insert MoveTo Path
memset(&path, 0, sizeof(path));
path.p1   = pos;
path.type = kMOVE_TO;
agent_paths.push_back(path);

for (int i = 0; i < kSIMULATION_DELAY && ok; i++) {
        // Next get agent
        ok = agent->RunOneUnit();
}

// reset agent
agent->Init(pos, angle);

for (int i = 0; i < steps; i++) {
        // Next get agent
        ok = agent->RunOneUnit();

        if (!ok) {
                break;
        }
        else {
                pos = agent->Pos();

                // Create path
                memset(&path, 0, sizeof(Path));
                path.type = kLINE_TO;
                path.p1 = pos;

                // Record path
                agent_paths.push_back(path);

                if (PointNear(pos, gPos, centre_width)) {
                        printf("FOUND FOOD in %d \n\n", i);
                        successfull_agents++;
                        found = true;
                        break;                  // fall out of loop
                }
        }
}

// End of Agent run
{
        // Insert EndTo Path
        memset(&path, 0, sizeof(path));
        path.type = kEND_PATH;
        agent_paths.push_back(path);
}
World::ReleaseAgent(agent);

// Copy successful agent's path to our drawing queue
if (found) {
```

```
                        Paths::iterator pos = agent_paths.begin();
                        for ( ; pos != agent_paths.end(); ++pos) {
                                World::AddPath(*pos);
                        }
                }
                else {
                        agent_paths.clear();
                }
        }

        // Add a legend
        std::ostringstream buf ;
        buf << num_agents << " agents: " << (distance/num_agents) << " vs " << sqrt(steps);
        std::string str = buf.str() ;
        World::AddLegend(str);

        return ok;
}



/*
 *  Random.h
 */
#pragma once

/**
 * Base class for random objects used in simulation
 * Uses rand() for number generation
 */
class Random
{
public:
        Random() {}
        virtual ~Random () {};

        /**
         * Seed the floating point generator
         * @note Must be called before Rand()
         * @see Rand
         * @param seed  The seed to use
         */
        virtual void    Seed (float seed);

        /**
         * Generates a random float in range [0.0, 1.0]
         * Uses rand.
         * @note rand generates ints in range [0, RAND_MAX] where RAND_MAX
         * is quite small. As you approach RAND_MAX sequence loses its randomness
         * @return A random float.
         */
        virtual float   Rand ();
};
```

```
/*
 *  Random.cpp
 */
#include <stdlib.h>
#include "Random.h"
#include "Utils.h"

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

```
void Random::Seed (float seed)
{
        SharedSrand();
}

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

float Random::Rand ()
{
        return ( (float)rand() / ((float) RAND_MAX + float(1)));
}
```

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

// Forward ref
class Node;
class Sensor;

/**
 * @class RandomAgent
 * A random simple agent based on N&Ps agent
 */
class RandomAgent : public Agent
{
public:
        /**
         * Default constructor
         */
        RandomAgent();

        /**
         * Destructor
         */
        ~RandomAgent();

        /**
         * Initialise the agent at a given point and orientation
         * @param       start   The starting location of the agent
         * @param       angle   The startint direction for the agent
         * @return True if agent is initialised, false otherwise
         */
        bool            Init            (const Point& start, 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      ();

private:
        Node*           m_left;                 //!< Random left motor node;
        Node*           m_right;                //!< Random right motor node;
        Node*           m_activation;   //!< Random activating node;
        Sensor*         m_sensor;               //!< Sensor
};
```

```
/*
```

```
 *  RandomAgent.cpp
 */

#include <math.h>

#include "Globals.h"
#include "Geo.h"
#include "Node.h"
#include "RandomNode.h"
#include "World.h"
#include "Agent.h"
#include "RandomAgent.h"
#include "Utils.h"
#include "Sensor.h"

/**
 * N&P state their agent moves 1 unit in each turn
 */
#define kDISTANCE_MOVED 1.0

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

RandomAgent::RandomAgent()  : m_activation(NULL)
{
        m_left   = new RandomNode();
        m_right  = new RandomNode();
        m_sensor = new Sensor();
        m_sensor->Attach(this);

        // Make connections
        m_sensor->ConnectTo(m_right, Node::Inhibitory);
        m_sensor->ConnectTo(m_left,  Node::Inhibitory);
}

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

RandomAgent::~RandomAgent()
{
        delete m_left;
        delete m_right;
        delete m_activation;
        delete m_sensor;
}

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

bool RandomAgent::Init (const Point& start, float angle)
{
        Agent::SetPos(start);
        Agent::SetAngle(angle);
        return true;
}

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

bool RandomAgent::RunOneUnit ()
{
        float l_value = m_left->CalcNodeValue();
        float r_value = m_right->CalcNodeValue();
        float s_value = m_sensor->CalcNodeValue();

        // Call sense

        // difference in
        // Calculate Turn position
```

```
        // N&P use the difference of the left and right values as
        // the turn angle (in radians)
        float tRad   = r_value - l_value;

        Agent::UpdatePosAngle(tRad, kDISTANCE_MOVED);

        // Update nodes
        m_left->SetNodeValue(l_value);
        m_right->SetNodeValue(r_value);
        m_sensor->SetNodeValue(s_value);

        return true;
}
```

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

/**
 * @class RandomMLCG
 *
 * A random number generator based upon Multiplicative Linear
 * Congruential Generator.
 * Generates random uniform variables in the range [0,1]
 */
class RandomMLCG : public Random
{
public:

        /**
         * Seed the random number generator
         * @param seed  The seed to use
         */
        void Seed (float seed);

        /**
         * Generate a random number
         * @returns A random number
         */
        float Rand ();
};
```

```
/*
 *  RandomMLCG.cpp
 */

#include <stdlib.h>
#include <Math.h>
#include <time.h>

#include "Random.h"
#include "RandomMLCG.h"
#include "Utils.h"

//---------------------------------------------------------------------
/*
 * Multplicative Linear Congruential Generator
```

```
 * (c) ??? Unsure have lost original reference
 * Better off using Mersene Twister code anyway.
 */
float MLCGRand()
{
        // 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;
}

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

void RandomMLCG::Seed (float seed)
{
        Random::Seed(seed);
}

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

float RandomMLCG::Rand ()
{
        return MLCGRand();
}
```

```
/*
 *  RandomMersenne.h
 */

// forward ref
class MTRand;

/**
 * @class RandomMersenne
 *
 * A random number generator based upon the Mersenne Twister
 * random number generation from Makoto Matsumoto et al
 * Generates random uniform variables in the range [0,1]
 */
class RandomMersenne : public Random
{
public:
        RandomMersenne();
        ~RandomMersenne();

        /**
         * Seed the random number generator
```

```
         * @param seed  The seed to use
         */
        void Seed (float seed);

        /**
         * Generate a random number
         * @returns A random number
         */
        float Rand ();

private:
        MTRand* m_rand;
};
```

```
/*
 *  RandomMersenne.cpp
 */

#include "MersenneTwister.h"
#include "Random.h"
#include "RandomMersenne.h"

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

RandomMersenne::RandomMersenne()
{
        // Create Mersenne Twister object
        m_rand = new MTRand();
}

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

RandomMersenne::~RandomMersenne()
{
        delete m_rand;
}

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

void RandomMersenne::Seed (float /* seed */)
{
        // do nothing, constructor of MTRand is self seeding
}

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

float RandomMersenne::Rand ()
{
         // MTRand rand() returns real number in [0,1]
        return (m_rand ? m_rand->rand() : 0.0);
}
```

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

// forward ref
class Random;
```

```
/**
 * @class RandomNode
 * A simple random node used as random oscillator
 */
class RandomNode : public Node
{
public:
        RandomNode();
        ~RandomNode();

        /**
         * [From Node Interface] Return the new node
         * value
         * @return The new value of this node
         */
        float   CalcNodeValue   () const;

private:
        Random* m_obj;
};
```

```
/*
 *  RandomNode.cpp
 */

#include "Random.h"
#include "RandomMLCG.h"
#include "Node.h"
#include "RandomNode.h"
#include <time.h>

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

RandomNode::RandomNode() : Node("Random") {
        m_obj = new RandomMLCG();
        m_obj->Seed(time(NULL));
}

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

RandomNode::~RandomNode()
{
        delete m_obj;
}

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

float RandomNode::CalcNodeValue () const
{
        return m_obj->Rand();
}
```

```
/*
 *  RandomNormal.h

 */
#pragma once

// forward ref
class MTRand;
```

```
/**
 * Generates random values with a normal/gaussian distribution
 */
class RandomNormal : public Random
{
public:
        /**
         *  Default constructor
         *      Has a mean = 0, and variance =1
         */
        RandomNormal();

        /**
         * Constructor
         *
         * @param mean          Mean of the distribution
         * @param variance      Variance of the distribution. Must be >= 0.0
         */
        RandomNormal(float mean, float variance);
        ~RandomNormal();

        /**
         * Generates a random float in range [0.0, 1.0] based on Box muleur method
         * to get normal/gaussian distribution
         * @return A random float.
         */
        float   Rand ();

private:
        float           m_mean;                         //!< Mean of normal distribution
        float           m_variance;                     //!< Variance of distribution
        float           m_pt;                           //!< Cached random value
        bool            m_cached;                       //!< True if we have a cached value
        float           m_stdDev;                       //!< Standard deviation of the distribution
        MTRand*         m_rand;                         //!< Uniform randon number generator object
};
```

```
/*
 *  RandomNormal.cpp
 */

#include "MersenneTwister.h"
#include "Random.h"
#include "RandomNormal.h"

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

RandomNormal::RandomNormal() : m_mean(0.0), m_variance(1.0),
m_cached(false), m_pt(0.0)
{
        m_rand   = new MTRand();
        m_stdDev = sqrt(m_variance);
}

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

RandomNormal::RandomNormal(float mean, float variance) :
m_mean(mean), m_variance(variance), m_cached(false), m_pt(0.0)
{
        // Can't have a negative variance
        assert(variance > 0.0);
```

```
        m_rand   = new MTRand();
        m_stdDev = sqrt(variance);
}

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

RandomNormal::~RandomNormal()
{
        delete m_rand;
}

//---------------------------------------------------------------------
// Uses box polar method to generate uniformly distributed points
// Adjusted version of method found in Numerical Recipes in C
float RandomNormal::Rand ()
{
        float x;

        if (m_cached) {
                m_cached = false;
                x        = m_pt;
        }
        else {
                float  w = 0.0;
                float v1, v2;

                do {
                        v1 = 2.0 * m_rand->rand() - 1.0;
                        v2 = 2.0 * m_rand->rand() - 1.0;
                        w  = (v1 * v1) + (v2 * v2);
                }
                while (w >= 1.0);

                // Following will give us normal distribution with
                // mean = 0, and variance = 1;
                w = sqrt( (-2.0 * log(w)) / w);
                x    = w * v1;
                m_pt     = w * v2;
                m_cached = true;               // // cache 2nd val for next time
        }
        // Adjust mean and variance
        return ( m_mean + m_stdDev * x);
}
```

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

// forward ref
class Agent;

/**
 * An odour sensor
 */
class Sensor : public Node
{
public:
        Sensor();
        ~Sensor();
```

```
        /**
         * Read from the environment
         */
        float   CalcNodeValue   () const;

        void    Attach (Agent* m_agent);

private:
        Agent*          m_agent;
};
```

```
/*
 *  Sensor.cpp
 */

#include "Globals.h"
#include "Geo.h"
#include "Node.h"
#include "Sensor.h"

#include "World.h"
#include "Agent.h"

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

Sensor::Sensor() : Node("Sensor"), m_agent(NULL)
{
}

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

Sensor::~Sensor()
{
}

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

float Sensor::CalcNodeValue     () const
{
        bool ok = false;
        float val = 0.0;
        if (m_agent) {
                 WorldInfo info;
                 info.data = 0.0;
                 ok = m_agent->Sense (info);
                 val = info.data;

                 if (val > 0.0) {
                        ok = true;                      // Debugging point
                 }
        }
        if (!ok) {
                val = Node::NodeValue();
        }
        return val;
}

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

void Sensor::Attach (Agent* agent)
{
        m_agent = agent;
}
```

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

class SignalGenerator
{
 public:
  //! Class constructor.
  SignalGenerator(  );

  //! Class destructor.
  virtual ~SignalGenerator(  );

  //! Return the last output value.
  virtual float Last( void ) const { return m_last; };

  //! Compute one sample and output.
  float Tick ( void );

 protected:

  // This abstract function must be implemented in all subclasses.
  // It is used to get around a C++ problem with overloaded virtual
  // functions.
  virtual float ComputeSample( void ) = 0;

  float m_last;
};
```

```
/*
 *  SignalGen.cpp
 */

#include "SignalGen.h"

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

SignalGenerator::SignalGenerator() : m_last(0.0)
{
}
//---------------------------------------------------------------------

SignalGenerator::~SignalGenerator( )
{
}

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

float SignalGenerator::Tick ( void )
{
        return ComputeSample();
}

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

float SignalGenerator::ComputeSample( void )
{
        return 0.0;
}
```

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

 #include <vector>

 // forward ref
 class Agent;

 /**
  * Minimal world to run agents within
  *
  * Offers two modes:
  *             Drawing paths
  *             or just plotting end points of run
  */
 class SimpleAgentWorld : public World
 {
 public:
        SimpleAgentWorld(Rect& rect);
        ~SimpleAgentWorld();

        //----- World Interface ------------------
        bool    Init    (void);
        bool    Run             (int num, int s);
        bool    Sense   (Point pos, WorldInfo& info);

private:
        /**
         * Internal call for run
         * Will plot the runs for n agents running
         * for t time.
         * @param       n       Number of agents to run
         * @param       t       Amount of time to run for
         * @return True if simulation was ok, false otherwise
         */
        bool VisualRun (int n, int t);

        /**
         * Internal call for run
         * Will plot the final point for n agents running
         * for t time.
         * @param       n       Number of agents to run
         * @param       t       Amount of time to run for
         * @return True if simulation was ok, false otherwise
         */
        bool VisualDot (int n, int t);
        Points          m_endPoints;    //!< Total distance covered by agents
 };
```

```
/*
 *  RandomWorld.cpp
 */

#if DEBUG
#include <iostream>
#include <fstream>
#endif

#include <sstream>
```

```
#include <string>
#include <vector>

#include "Globals.h"
#include "math.h"
#include "Geo.h"
#include "Node.h"
#include "World.h"
#include "Agent.h"
#include "Random.h"
#include "RandomAgent.h"
#include "SimpleAgentWorld.h"
#include "Utils.h"

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

SimpleAgentWorld::SimpleAgentWorld(Rect& rect) : World(rect)
{
}

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

SimpleAgentWorld::~SimpleAgentWorld()
{
}

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

bool SimpleAgentWorld::Init ()
{
        return true;
}

//-----------------------------------------------------
// Just defer to default world
bool SimpleAgentWorld::Sense    (Point pos, WorldInfo& info){
        return World::Sense(pos, info);
}

//-----------------------------------------------------
// Logs full path of agents
bool SimpleAgentWorld::VisualRun (int num_agents, int steps)
{
        // Grab a random number generator
        Random rand;

        bool  ok       = true;
        float distance = 0.0;

        for (int num = 0; num < num_agents; num++) {

                // Start agent in middle
                float x         = 0.0;
                float y         = 0.0;

                Point pos   = {x, y};
                float angle = rand.Rand() * 360.0; // generate random angle

                // Add new agent
                Agent * agent = World::GetAgent();

                if (agent == NULL)
                        break;

                agent->Init(pos, angle);
```

```
                agent->Embed(this);

                // Insert MoveTo Path
                Path path;
                memset(&path, 0, sizeof(path));
                path.p1   = pos;
                path.type = kMOVE_TO;
                World::AddPath(path);

                for (int i = 0; i < kSIMULATION_DELAY && ok; i++) {
                        // Next get agent
                        ok = agent->RunOneUnit();
                }

                // reset agent
                agent->Init(pos, angle);

                for (int i = 0; i < steps; i++) {
                        // Next get agent
                        ok = agent->RunOneUnit();

                        if (!ok) {
                                break;
                        }
                        else {
                                pos = agent->Pos();

                                // Create path
                                Path path;
                                memset(&path, 0, sizeof(Path));
                                path.type = kLINE_TO;
                                path.p1 = pos;

                                // Record path
                                World::AddPath(path);
                        }
                }

                // End of Agent run
                {
                        // Insert EndTo Path
                        memset(&path, 0, sizeof(path));
                        path.type = kEND_PATH;
                        World::AddPath(path);
                }
                World::ReleaseAgent(agent);
        }

        // Add a legend
        std::ostringstream buf ;
        buf << num_agents << " agents: " << (distance/num_agents) << " vs " << sqrt(steps);
        std::string str = buf.str() ;
        World::AddLegend(str);

        return ok;
}


//-----------------------------------------------------
// Logs just end point
bool SimpleAgentWorld::VisualDot (int num_agents, int steps)
{
        // Grab a random number generator
        Random rand;
```

```
bool  ok       = true;

// set up mean squares
double mean_squares[steps];

for (int j = 0; j < steps; j++) {
        mean_squares[j] = 0.0;
}

for (int num = 0; num < num_agents; num++) {

        // Start agent in middle
        float x         = 0.0;
        float y         = 0.0;

        Point pos   = {x, y};
        float angle = rand.Rand() * 360.0; // generate random angle

        // Add new random agent
        // Add new agent
        Agent * agent = World::GetAgent();
        if (agent == NULL)
                break;

        agent->Init(pos, angle);

        // Insert MoveTo Path
        Path path;
        memset(&path, 0, sizeof(path));
        path.p1   = pos;
        path.type = kMOVE_TO;
        World::AddPath(path);

        for (int i = 0; i < steps; i++) {
                ok = agent->RunOneUnit();

                if (!ok) {
                        break;
                }
                else {
                        pos = agent->Pos();

                        // Add to mean squares
                        mean_squares[i] += (double) ((pos.x * pos.x) + (pos.y * pos.y));
                }
        }

        // End of Agent run
        {
                // Create path
                Path path;
                memset(&path, 0, sizeof(Path));
                path.type = kPOINT_TO;
                path.p1 = pos;

                // Record path
                World::AddPath(path);

                // Insert EndTo Path
                memset(&path, 0, sizeof(path));
                path.type = kEND_PATH;
                World::AddPath(path);
        }
        World::ReleaseAgent(agent);
}
```

```
        // finally calc mean squares and log to file
        std::ofstream stm;
        stm.open("/Users/pip/Desktop/Meansquare.dat");

        stm << "#plot \"Meansquare.dat\" using 1:2 title 'Right' with lines, \
                plot \"Meansquare.dat\" using 1:3 title 'uniform'" << std::endl;
        stm << "#step\tMSD" << std::endl;
        for (int i = 0; i < steps; i ++) {
                mean_squares[i] = mean_squares[i] / (double) num_agents;
                // Times sequence is from [1:steps] so add one to time value
                // Log the timestep twice so we have a source for log(time) = log(msd)
                stm << log(i+1) << "\t" << log(mean_squares[i]) << "\t" << log(i+1) << std::endl;
        }
        stm.close();

        return ok;
}

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

bool SimpleAgentWorld::Run (int n, int steps)
{
        return VisualDot (n, steps);
}
```

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

// forward ref
class Sensor;

/**
 * @class SineAgent
 * A simple agent based on N&Ps agent using
 * noisey sine wave oscillators to generate motion
 */
class SineAgent : public Agent
{
public:
        /**
         * Default constructor
         */
        SineAgent();

        /**
         * Destructor
         */
        ~SineAgent();

        /**
         * Initialise the agent at a given point and orientation
         * @param       start   The starting location of the agent
         * @param       angle   The startint direction for the agent
         * @return True if agent is initialised, false otherwise
         */
        bool            Init            (const Point& start, 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      ();

private:
        Node*           m_left;                 //!< Left motor node;
        Node*           m_right;                //!< Right motor node;
        Node*           m_activation;   //!< Activating node;
        Sensor*         m_sensor;
};
```

```
/*
 *  SineAgent.cpp
 */

/**
 * If true log agent values
 */
#define AGENT_LOGGING 1

/**
 * Log agent init values
 */
#define DISTANCE_LOGGING 0

#if AGENT_LOGGING || DISTANCE_LOGGING
#include <iostream>
#include <fstream>
#endif

#include <math.h>

#include "Globals.h"
#include "Geo.h"
#include "Node.h"
#include "SineNode.h"
#include "World.h"
#include "Agent.h"
#include "SineAgent.h"
#include "Utils.h"
#include "Random.h"
#include "Sensor.h"
/*
 * Allow left and right turn motors to go negative
 */
#define kOSCILLATOR_MIN_VALUE -1.0

/*
 * Activating oscillator can only be positive
 */
#define kACTIVATING_MIN_VALUE 0.001

/*
 * N&P define a the max value for an oscillator to be 1.0
 */
#define kOSCILLATOR_MAX_VALUE 1.0


#if DISTANCE_LOGGING
std::ofstream distance;
static float max_distance = 0.0;
#endif // DISTANCE_LOGGING
```

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

#if AGENT_LOGGING
std::ofstream leftstream;
std::ofstream rightstream;
std::ofstream activationstream;
std::ofstream turnStream;
std::ofstream sensStream;
static int stepcount = 0;

static void OpenLogs (void);
static void CloseLogs (void);
#endif  // kAGENT_LOGGING


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

SineAgent::SineAgent()  : m_activation(NULL)
{

#if DISTANCE_LOGGING
        distance.open("/Users/pip/Desktop/Agent.txt", std::ios::app);
        Random randObj;

        // TODO: Evolve parameters to see if its possible
        // to generate an agent using sine waves
        float freq = randObj.Rand() * 10.0;
        float amp  = randObj.Rand();
        float phase = randObj.Rand();
        m_left                  = new SineNode(amp, phase, freq);

        distance << "L: " << amp << " " << phase << " " << freq;

        phase = randObj.Rand();
        m_right                 = new SineNode(amp, phase, freq);

        distance << "   R: " << amp << " " << phase << " " << freq;

        freq  = randObj.Rand() * 10.0;
        phase = randObj.Rand();
        amp  = randObj.Rand();
        m_activation    = new SineNode(amp, phase, freq);

        distance << "    A: " << amp << " " << phase << " " << freq << std::endl;

        distance.close();
#else
        // Best values from 1000 agents (amp, phase, frequency)
        // L: 0.487784 0.18513 0.516147   R: 0.487784 0.483748 0.516147
        // A: 0.341732 0.586681 3.57564

        const float turnFreq = 0.516;
        const float turnAmp  = 0.488;
        const float actAmp   = 0.342;
        const float actFreq  = 3.576;

        m_left                  = new SineNode(turnAmp, 0.185, turnFreq);
        m_right                 = new SineNode(turnAmp, 0.484, turnFreq);
        m_activation    = new SineNode(actAmp,  0.587, actFreq);
        m_sensor                = new Sensor();

#endif // DISTANCE_LOGGING

        m_left->ConnectTo(m_right, Node::Inhibitory);
```

```
        m_right->ConnectTo(m_left, Node::Inhibitory);

        m_activation->ConnectTo(m_right, Node::Excitatory);
        m_activation->ConnectTo(m_left,  Node::Excitatory);

        m_sensor->Attach(this);
        m_sensor->ConnectTo(m_right, Node::Inhibitory);
        m_sensor->ConnectTo(m_left, Node::Inhibitory);

#if AGENT_LOGGING
        OpenLogs();
#endif
}

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

SineAgent::~SineAgent()
{

#if DISTANCE_LOGGING
        Point pos = Pos();
        float dist = sqrt( (pos.x * pos.x) + (pos.y * pos.y) );
        if (dist > max_distance) {
                max_distance = dist;

                distance.open("/Users/pip/Desktop/Agent.txt", std::ios::app);
                distance << "-----> Dist: " << dist << std::endl << std::endl;
                distance.close();
        }
#endif
        delete m_left;
        delete m_right;
        delete m_activation;
        delete m_sensor;

#if AGENT_LOGGING
        CloseLogs();
#endif
}

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

bool SineAgent::Init (const Point& start, float angle)
{
        Agent::SetPos(start);
        Agent::SetAngle(angle);
        return true;
}

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

float Normalise (float val, float min_val, float max_val)
{
        if (val < min_val) {
                val = min_val;
        }
        else if (val > max_val) {
                val = max_val;
        }
        return val;
}

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

bool SineAgent::RunOneUnit ()
```

```
{
        float l_value = Normalise(m_left->CalcNodeValue(),
                                          kOSCILLATOR_MIN_VALUE, kOSCILLATOR_MAX_VALUE);
        float r_value = Normalise(m_right->CalcNodeValue(),
                                          kOSCILLATOR_MIN_VALUE, kOSCILLATOR_MAX_VALUE);
        float a_value = Normalise(m_activation->CalcNodeValue(),
                                          kACTIVATING_MIN_VALUE, kOSCILLATOR_MAX_VALUE);
        float s_value = m_sensor->CalcNodeValue();

        // difference in
        // Calculate Turn position
        // N&P use the difference of the left and right values as
        // the turn angle (in radians)
        float tRad   = r_value - l_value;
        Agent::UpdatePosAngle(tRad, kDISTANCE_MOVED);

        // Update nodes
        m_left->SetNodeValue(l_value);
        m_right->SetNodeValue(r_value);
        m_activation->SetNodeValue(a_value);
        m_sensor->SetNodeValue(s_value);

#if AGENT_LOGGING
        leftstream        << stepcount << "\t" << l_value << std::endl;
        rightstream       << stepcount << "\t" << r_value << std::endl;
        activationstream  << stepcount << "\t" << a_value << std::endl;
        turnStream        << stepcount << "\t" << tRad    << std::endl;
        sensStream        << stepcount << "\t" << s_value << std::endl;
        stepcount++;
#endif
        return true;
}



//---------------------------------------------------------------------
// Log utility code
//---------------------------------------------------------------------
#if AGENT_LOGGING

void OpenLogs ()
{
        // reset counter
        stepcount = 0;

        // Open left oscillator log
        std::string buffer = kDEFAULT_FILE_PATH;
        buffer += "LeftTurn.dat";
        leftstream.open(buffer.c_str());
        leftstream << "#plot \"LeftTurn.data\" using 1:2 title 'Left' with lines" << std::endl;
        leftstream << "#Step\tLeft" << std::endl;

        // Open right oscillator log
        buffer = kDEFAULT_FILE_PATH;
        buffer += "RightTurn.dat";
        rightstream.open(buffer.c_str());
        rightstream << "#plot \"RightTurn.data\" using 1:2 title 'Right' with lines" << std::endl;
        rightstream << "#Step\tRight" << std::endl;

        // Open activating oscillator log
        buffer = kDEFAULT_FILE_PATH;
        buffer += "Activation.dat";
        activationstream.open(buffer.c_str());
        activationstream << "#plot \"Activation.data\" using 1:2 title 'Activation' with lines" << std::endl;
        activationstream << "#Step\tActivation" << std::endl;
```

```
        // Open turn angle log
        buffer = kDEFAULT_FILE_PATH;
        buffer += "TurnAngle.dat";
        turnStream.open(buffer.c_str());
        turnStream << "#plot \"TurnAngle.data\" using 1:2 title 'Angle' with lines" << std::endl;
        turnStream << "#Step\tAngle" << std::endl;

        // Open sensor log
        buffer = kDEFAULT_FILE_PATH;
        buffer += "SenseData.dat";
        sensStream.open(buffer.c_str());
        sensStream << "#plot \"SenseData.dat\" using 1:2 title 'Sense' with lines" << std::endl;
        sensStream << "#Step\tSense" << std::endl;
}

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

void CloseLogs ()
{
        leftstream.close();
        rightstream.close();
        activationstream.close();
        turnStream.close();
        sensStream.close();
}

#endif
```

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


#include <vector>

typedef std::vector<float>              FloatArray;

/**
 * Generates a sampled sine wave
 */
class SineGenerator : public SignalGenerator
{
public:
        /**
         * Number of samples to generate
         * Default amp = 1.0, phase = 0.0 and frequency = 1 rad/s
         */
        SineGenerator(int num_samples);

        /**
         * Constructor
         * Generates a sine singal with a custom amplitude, frequency and phase
         * @param num_samples   The number of samples to generate
         * @param amp                   Peak amplitude of the signal
         * @param freq                  The frequency of the signal in rad/s
         * @param phase                 The phase angle of the signal (shifts signal in time)
         */
        SineGenerator(int num_samples, float amp, float freq, float phase);
        ~SineGenerator();
```

```
        float Tick ( void );

 protected:

        // This abstract function must be implemented in all subclasses.
        // It is used to get around a C++ problem with overloaded virtual
        // functions.
        float ComputeSample( void );


private:
        FloatArray      m_samples;
        int                     m_time;
        int                     m_maxTime;
        float           m_amp;
        float           m_phase;
        float           m_freq;
};
```

```
/*
 *  SineGen.cpp
 */
#include <math.h>
#include "SignalGen.h"
#include "SineGen.h"

static void Init (FloatArray& v, int num_samples, float phase, float amp, float freq);

//---------------------------------------------------------------------
// Generate samples for sinusoidal wave
void Init (FloatArray& v, int num_samples, float phase, float amp, float freq)
{
        const double PI = 4 * atan(1.0);

        v.reserve(num_samples);

        // Convert frequency to angular frequency
        float w = freq * 2 * PI;

        for (int n = 0; n < num_samples; n++ ) {
                float val = amp * sin( ((w * n) / num_samples) + phase);
                v.push_back(val);
        }
}

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

SineGenerator::SineGenerator(int num_samples) :
m_time(0), m_maxTime(num_samples), m_amp(1.0), m_phase(0.0), m_freq (1.0)
{
        Init(m_samples, num_samples, m_phase, m_amp, m_freq);
}

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

SineGenerator::SineGenerator(int num_samples, float amp, float freq, float phase) :
m_time(0), m_maxTime(num_samples), m_amp(amp), m_phase(phase), m_freq(freq)
{
        Init(m_samples, num_samples, m_phase, m_amp, m_freq);
}

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

```
SineGenerator::~SineGenerator()
{
}

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

float SineGenerator::Tick ( void )
{
        return ComputeSample();
}

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

float SineGenerator::ComputeSample( void )
{
        float sample = m_samples.at(m_time++);

        // Wrap around
        if (m_time >= m_maxTime)
                m_time = 0;

        return sample;
}
```

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

// forward ref
class SineGenerator;
class Random;

/**
 * @class SineNode
 * A node used based on a sine oscillator
 */
class SineNode : public Node
{
public:
        /**
         * Constructor
         */
        SineNode(float amp, float phase, float freq);
        ~SineNode();

        /**
         * [From Node Interface] Return the new node
         * value
         * @return The new value of this node
         */
        float   CalcNodeValue   () const;

private:
        SineGenerator*  m_obj;
        Random *                m_noise;
};
```

```
/*
```

```
 *  SineNode.cpp
 */

#include "Globals.h"

#include "SignalGen.h"
#include "SineGen.h"
#include "Node.h"
#include "SineNode.h"
#include "Random.h"
#include "RandomNormal.h"

/**
 * Amount to step down noise signal
 */
#define kNOISE_MODIFIER 1.0

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

SineNode::SineNode(float amp, float phase, float freq) : Node("SineNode")
{
        m_obj   = new SineGenerator(kSIMULATION_STEPS, amp, phase, freq);
        m_noise = new RandomNormal();
}

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

SineNode::~SineNode()
{
        delete m_obj;
        delete m_noise;
}

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

float SineNode::CalcNodeValue   () const
{
        // Cast away const ness of random object
        Random* r_ptr = const_cast<Random*>(m_noise);

        float val = m_obj->Tick();
        // add just a small amount of noise
        val += (r_ptr->Rand() / kNOISE_MODIFIER);
        return val;
}
```

```
/*
 *  Utils.h
 *  Place holder for shared code and utility functions and classes.
 */

#pragma once


/**
 * @class Transform
 * Simple interface class used to transform one numeric value to another
 */
template <class T>
class Transform {
public:
```

```
        virtual T Op(T x) = 0;
};

/**
 * @class NormaliseMin
 * Simple checking rule used to keep values in bound
 */
template <class T>
class NormaliseMin  : public Transform<T> {
public:
        NormaliseMin(T min) : m_min(min) {}
        T Op (T x)                      {return (x < m_min ? m_min : x);}
private:
        T m_min;
};

/**
 * @class NormaliseMax
 * Simple checking rule used to keep values in bound
 */
template <class T> class NormaliseMax : public Transform<T> {
public:
        NormaliseMax(T max) : m_max(max) {}
        T Op (T x)                       {return (x > m_max ? m_max : x);}
private:
        T m_max;
};

/**
 * @class NormaliseMaxMin
 * Simple checking rule used to keep values in bound
 */
template <class T> class NormaliseMaxMin : public Transform<T> {
public:
        NormaliseMaxMin(T max, T min) : m_max(max), m_min(min) {}
        T Op (T x)                       {
                if (x > m_max)
                        x = m_max;
                else if (x < m_min)
                        x = m_min;
                return x;
        }
private:
        T m_max;
        T m_min;
};

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

double  RadiansToDegrees        (double rad);
float   NormaliseAngle          (float angle);

/**
 * Shared call to SRand
 * Issues seeding only happens once
 */
void    SharedSrand                     ();

/**
 * Checks that value is valid
 */
inline bool IsNan (double x)
{
        return x != x;
}
```

```
/*
 *  Utils.cpp
 */

#include <stdlib.h>
#include <time.h>

#include "Globals.h"
#include "Utils.h"

// Global - Remembers if srand has been called

static bool srand_init = false;

//--------------------------------------------------------------------
// convert radians to degrees
double RadiansToDegrees (double rad)
{
    return rad * (180.0 / kPI);
}

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

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

        while (angle < 0.0) {
                angle += 360.0;
        }
        return angle;
}


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

void SharedSrand ()
{
        if (srand_init == false) {
                srand_init = true;

                srand(time(NULL));
        }
}
```

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

#include <vector>
#include <string>

// Forward ref
class Agent;

/**
 * Information about the world at a given point
 * For now data == odour gradient
```

```
 */
typedef struct {
        float   data;                   //!< data in the world
        size_t  size;                   //!< Size of the data structure
}
WorldInfo;

/**
 * Callback to add an agent to the world
 */
typedef Agent* (*AgentProc) (void *clientData);

/**
 * @class World
 * Abstract interface for simulation world
 */
class World
{
        typedef std::vector<std::string>        Legends;

public:
        World(Rect bounds);
        virtual ~World();

        /**
         * Register a callback to create agents for the world
         * @param proc  The callback to use to create an agent
         * @param data  The data to pass with the callback
         * @return True if callback was registered, false otherwise
         */
        virtual bool    AddAgentCB (AgentProc proc, void *data);

        /**
         * Init the world
         * @return True if the world was initialised, false otherwise
         */
        virtual bool    Init       () = 0;

        /**
         * Run the simulation for num agents running s simulation steps
         * @param num   The number of agents to simulate
         * @param s             The number of simulation steps
         * @return      True if simulation was ok, false otherwise
         */
        virtual bool    Run  (int num, int s) = 0;

        /**
         * Switches display from showing data to showing
         * paths.
         * @param data If true, show data, otherwise show paths
         */
        virtual void ToggleData (bool data);

        // Draw World
        static void     Display ();
        static void Draw ();

        /**
         * Request information about the world at a given point
         * @param pos   The position in the world
         * @param info  Information about that position
         * @return      True if the world was sampled correctly
         */
        virtual bool    Sense   (Point pos, WorldInfo& info) = 0;
```

```
protected:
        /**
         * Convenience function
         * Uses registed AgentProc to ask for an agent
         */
        Agent*  GetAgent ();

        /**
         * Convenience function
         * Release an agent when its finished with
         */
        void    ReleaseAgent (Agent*& agent);


        /**
         * Add a path to the drawing queue
         * @param path  The path object to draw
         */
        void    AddPath         (Path path);

        /**
         * Add a lengend string to the drawing queue
         * @param message       The text to draw
         */
        void    AddLegend       (std::string message);

        static Paths            m_paths;                //!< Shared memory for all worlds
        static Legends          m_legend;               //!< Strings to show as legends
        static Point            m_max;                  //!< Largest seen x,y co-ords (assumes square world)
        static bool                     m_data;                 //!< If true, show data

        Rect                            m_rect;                 //!< Bounds of the world
//      Agent*                          m_agent;

        AgentProc                       m_proc;
        void*                           m_procData;

private:
        World(const World& world) {};
};
```

```
/*
 *  World.cpp
 */
#include <vector>
using namespace std;

#include <math.h>
#include <GLUT/glut.h>

#include "Globals.h"
#include "Geo.h"
#include "World.h"
#include "Agent.h"
#include "miGLUtils.h"

// List of geometric paths to draw
Paths                   World::m_paths;
World::Legends  World::m_legend;
Point                   World::m_max = {0.0, 0.0};
bool                    World::m_data = false;
```

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

World::World(Rect bounds) :
m_rect(bounds), m_proc(NULL), m_procData(NULL)
{
}

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

World::~World()
{
}

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

void World::ToggleData (bool data)
{
        m_data = data;
}

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

bool World::Init ()
{
        m_paths.clear();
        return true;
}

//---------------------------------------------------------------------
// TODO: Add safety mechanism to ensure memory leaks/unsafe behaviour
// doesn't occur when using data
bool World::AddAgentCB (AgentProc proc, void * data)
{
        m_proc = proc;
        m_procData = data;
        return true;
}

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

void World::AddPath (Path path)
{
        // Add line object
        m_paths.push_back(path);

        if ( (path.type == kLINE_TO) || (path.type == kPOINT_TO) ) {
                if (fabs(path.p1.x) > m_max.x) {
                        m_max.x = fabs(path.p1.x);
                }
                if (fabs(path.p1.y) > m_max.y) {
                        m_max.y = fabs(path.p1.y);
                }
        }
        else if (path.type == kCIRCLE_PATH) {
                float radius = fabs(path.p2.x);
                if ( (fabs(path.p1.x) + radius) > m_max.x) {
                        m_max.x = fabs(path.p1.x + radius);
                }
                if ( (fabs(path.p1.y) + radius) > m_max.y) {
                        m_max.y = fabs(path.p1.y + radius);
                }
        }

}
```

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

void World::AddLegend (std::string message)
{
        m_legend.push_back(message);
}

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

Agent* World::GetAgent ()
{
        Agent * agent = NULL;
        if (m_proc) {
                try{
                        agent = (m_proc) (m_procData);
                }
                catch (...) {
                }
        }
        return agent;
}

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

void World::ReleaseAgent (Agent*& agent)
{
        delete agent;
        agent = NULL;
}

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

void World::Display ()
{
        glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(1.0, 1.0, 1.0);
        glLoadIdentity();

        Draw();
        glutSwapBuffers();
}

//---------------------------------------------------------------------
// TODO: replace with a lookup table
void SetRandomColour(int &batton)
{
        // TODO: use glEnable(GL_LINE_STIPPLE)
        // & glDisable (GL_LINE_STIPPLE) to switch to dashed lines etc
        if (batton > 9)
                batton = 0;

        switch (batton) {
                // black
                case 0:
                        glColor3f(0.0, 0.0, 0.0);
                        break;

                // red
                case 1:
                        glColor3f(1.0, 0.0, 0.0);
                        break;

                // green
                case 2:
```

```
                        glColor3f(0.0, 1.0, 0.0);
                        break;

                // blue
                case 3:
                        glColor3f(0.0, 0.0, 1.0);
                        break;

                // orange
                case 4:
                        glColor3f(1.0f, 0.6f, 0.0);
                        break;

                // Magenta
                case 5:
                        glColor3f(1.0, 0.0, 1.0);
                        break;

                // Yellow
                case 6:
                        glColor3f(1.0f, 1.0f, 0.0f);
                        break;

                // Cyan
                case 7:
                        glColor3f(0.0f, 1.0f, 1.0f);
                        break;

                // Grey
                case 8:
                        glColor3f(0.5, 0.5, 0.5);
                        break;

                // Hot pink
                case 9:
                        glColor3f(1.0f, 0.4f, 0.7f);
                        break;
        }
        batton++;
}

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

void World::Draw ()
{
        // Set Clear colour to white
        glClearColor(1.0, 1.0, 1.0, 0.0);
        glClear(GL_COLOR_BUFFER_BIT);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        GLdouble max_point = (m_max.x > m_max.y ? m_max.x : m_max.y);
        GLdouble max_dim  = max_point + (max_point * 0.4);              // add 40%

        // Centre co-ordinates at 0.0 based on our GL window size
        // Scale point system to match pixels, maintaing aspect ratio
        GLdouble ratio  = (GLfloat) kWindowHeight / (GLfloat)kWindowWidth;
        GLdouble bottom =  -max_dim * ratio;
        GLdouble top    =   max_dim * ratio;
        glOrtho(-max_dim, max_dim, bottom, top, -1.0, 1.0);

        // Set the drawing color to grey
        glColor3f(0.5, 0.5, 0.5);

        // Draw a cube around the start position
```

```
const float cube_half = 1.0;
glBegin(GL_POLYGON);
        // Specify verticies in quad
        glVertex2f(-cube_half, -cube_half);
        glVertex2f(-cube_half, cube_half);
        glVertex2f(cube_half,  cube_half);
        glVertex2f(cube_half,  -cube_half);
glEnd();

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

Point currentPoint = {0.0, 0.0};
int colours = 0;
int agents  = 0;
bool open_path = false;

// Iterator over all cached path objects and draw on screen
// Each MoveTo/End_Path pair to signals a new agent path so
// handle accordingly.
for (Paths::iterator pos = m_paths.begin(); pos != m_paths.end(); ++pos) {
        Path path = (*pos);

        switch (path.type) {
                case kLINE_TO:
                        {
                                DrawLine(currentPoint, path.p1);
                                currentPoint = path.p1;         // update current path
                        }
                        break;

                case kPOINT_TO:
                        {
                                open_path = true;
                                currentPoint = path.p1;

                                glPointSize(2.0);
                                // Draw point
                                glBegin(GL_POINTS);
                                        glVertex2d(path.p1.x, path.p1.y);
                                glEnd();
                                glPointSize(1.0);
                        }
                        break;

                case kMOVE_TO:
                        {
                                open_path = true;
                                agents++;                               // cheat: see only one move to per agent
                                currentPoint = path.p1;
                                glRasterPos2f(path.p1.x, path.p1.y);
                                SetRandomColour(colours);
                        }
                        break;

                case kCIRCLE_PATH:
                {
                        DrawLineCircle(path.p2.x, path.p1.x, path.p1.y);
                        currentPoint = path.p1;
                }
                break;

                case kEND_PATH:
                        {
                                if (open_path) {
```

```
                                        //char buffer[100] = "\0";
                                        //sprintf(buffer, "%d", agents);
                                        //DrawText(buffer, currentPoint.x, currentPoint.y, 10);
                                        glColor3f(0.5, 0.5, 0.5);
                                        DrawFilledCircle(1.0, currentPoint.x, currentPoint.y);
                                }
                                open_path = false;
                        }
                        break;
                }
        }

        // Now draw legend
        DrawScaleBars(-max_dim, max_dim, bottom);

        // x point just inside left boundary
#if 0
        float x = -kSIMULATION_STEPS + 5.0;
        float y = (float) top - 20.0;
        Point legendPoint = {x, y};
        for (Legends::iterator pos = m_legend.begin();
                pos != m_legend.end(); ++pos) {
                std::string message = (*pos);
                DrawText( message.c_str(), legendPoint.x, legendPoint.y, 10);

                legendPoint.y -= 10;
        }
#endif


        // Flush the buffer to force drawing of all objects thus far
        glFlush();
}

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

```
/*
 * Main.cpp
 *
 */
#include <string>
#include <iostream>
#include <GLUT/glut.h>

#include "Globals.h"
#include "Geo.h"
#include "World.h"
#include "Agent.h"
#include "SimpleAgentWorld.h"
#include "OdourWorld.h"
#include "GLToJPEG.h"

#include "RandomAgent.h"
#include "NPAgent.h"
#include "SineAgent.h"

#define NPAGENT 0

//------------------------------------------------------------------------
// Global world object
World* gWorld = NULL;

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

```
/*
 * Code for the Right Click Menu
 */
void Glut_Menu (int id)
{
        // Examine the request
        switch (id) {
                case 1:
                {
                        gWorld->ToggleData(true);
                }
                break;

                case 2:
                {
                        gWorld->ToggleData(false);
                }
                break;

                case 3:
                {
                        // 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 4:
                {
                        // User has asked to quit
                        exit(0);
                }
                break;
        }
}


//----------------------------------------------------------------------
// 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("Show data", 1);
        glutAddMenuEntry("Show paths", 2);
        glutAddMenuEntry("Export as JPEG", 3);
```

```
        glutAddMenuEntry("Quit", 4);
        glutAttachMenu(GLUT_RIGHT_BUTTON);

        return id;
}

//------------------------------------------------------------------------
// Callback to create random agents in the world
Agent* RandomAgentCall (void * data)
{
        RandomAgent * agent = new RandomAgent();
        return agent;
}

//------------------------------------------------------------------------
// Callback to create logistic agents in the world
Agent* NPAgentCall (void * data)
{
        NPAgent * agent = new NPAgent();
        return agent;
}

//------------------------------------------------------------------------
// Create sine agents in the world
Agent* SineAgentCall (void * data)
{
        SineAgent * agent = new SineAgent();
        return agent;
}

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

int main(int argc, char** argv)
{
        // Drawing at end
        Rect bounds = {0, 0, kWindowWidth, kWindowHeight};

        // ID should be positive
        int id = InitGLUT(argc, argv, "Wandering");

#if WANDER
        // Random agent and world
        // Run Agent
        gWorld = new SimpleAgentWorld(bounds);
        gWorld->AddAgentCB(NPAgentCall, NULL);

        gWorld->Init();
        gWorld->Run(100, kSIMULATION_STEPS);
#else
        gWorld = new OdourWorld(bounds);
        gWorld->AddAgentCB(NPAgentCall, NULL);

        gWorld->Init();
        gWorld->Run(1, kSIMULATION_STEPS);
#endif

        // Get Results & translate into OpenGL commands to show Agent's path
        if (id >= 1) {
                glutDisplayFunc(World::Display);

                glutMainLoop();
        }
        delete gWorld;

        return 0;
```

```
}
```

```
/*
 *  GLUtils.h

 */

#define KDEFAULT_FONT_SIZE 10

/**
 * Draws a circle outline
 * @param radius        The radius of the circle
 * @param x                     The x position of the centre
 * @param y                     The y position of the centre
 */
void DrawLineCircle (float radius, float x, float y);

/**
 * Draws a filled circle
 * @param radius        The radius of the circle
 * @param x                     The x position of the centre
 * @param y                     The y position of the centre
 */
void DrawFilledCircle (float radius, float x, float y);

/**
 * Draws a text buffer in a bitmap helvetica font.
 * @param string        The text buffer to draw
 * @param x                     The start x position of the text
 * @param y                     The start y position of the text
 * @param size          [Optional] The size of the text (10, 12, 18 points)
 */
void DrawText(const char *string, float x, float y, int size = KDEFAULT_FONT_SIZE);

/**
 * Draws a line
 * @param start         The start position of the line
 * @param end           The end position of the line
 */
void DrawLine (Point start, Point end);


/**
 * Draws a scale bar in the bottom left hand corner
 * @param left          The leftmost position of the window
 * @param right         The rightmost position of the window
 * @param bottom        The bottomost position of the window
 */
void DrawScaleBars (float left, float right, float bottom);
```

```
/*
 *  GLUtils.cpp
 */
#include <math.h>
#include <GLUT/glut.h>

#include "Globals.h"
#include "Geo.h"
#include "miGLUtils.h"
```

```
//-----------------------------------------------------------------
// Convenience value used to convert degrees to radians
const float DEG2RAD             = kPI/180;

//-----------------------------------------------------------------
/**
 * Draws a 'line' circle of radius 'rad' and position x,y)
 */
void DrawLineCircle (float radius, float x, float y)
{
        glBegin(GL_LINE_LOOP);

        for (int i=0; i < 360; i++) {
                float degInRad = i * DEG2RAD;
                glVertex2f((cos(degInRad)*radius) + x, (sin(degInRad)*radius) + y);
        }
        glEnd();
}

//-----------------------------------------------------------------
/*
 * Modified version of code from Glen E. Gardner Jr
 * (c) 1998
 * http://www.gmonline.demon.co.uk/cscene/CS6/CS6-06.html
 */
void DrawFilledCircle (float radius, float x, float y)
{
        float x1 = x;
        float v1 = y;
        float angle = 0.0;

        glBegin(GL_TRIANGLES);

        for(int i=0; i <= 360; i++) {
                // angle = (float)(((double)i) / 57.29577957795135);
                angle = i * DEG2RAD;
                float vx = x + (radius * (float)sin((double)angle));
                float vy = y + (radius * (float)cos((double)angle));
                glVertex2d(x, y);
                glVertex2d(x1, v1);
                glVertex2d(vx, vy);
                v1 = vy;
                x1 = vx;
        }
        glEnd();
}

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

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

```
        }

        int len = (int) strlen(string);
        for (int i = 0; i < len; i++) {
                // Draw Text
                glutBitmapCharacter(font, string[i]);
        }
}

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

void DrawLine (Point start, Point end)
{
        glBegin(GL_LINES);
                glVertex2f(start.x , start.y);
                glVertex2f(end.x,    end.y);
        glEnd();
}


//-----------------------------------------------------------------
// Draws scale bars relative to width of window
void DrawScaleBars (float left, float right, float bottom)
{
        // Switch to black ink
        glColor3f(0.0, 0.0, 0.0);

        float mag = 1.0;

        // First work out order of magnitude of width
        float width = right - left;
        width /= 10.0;
        while (width > 1.0) {
                width /= 10.0;
                mag *= 10.0;
        }

        const float len   = 1 * mag;            // Length is just order of magnitude for window
        const float unit  = len / 5.0;          // Chop into five subsections
        const float hunit = unit / 4.0;         // Relative length of vertical bar

        Point start, end;

        // Add a small value to start values to get
        // away from window edge
        left   += unit;
        bottom += unit;

        // Draw main horizontal bar
        start.x = left; start.y = bottom;
        end.x   = left + len; end. y = bottom;
        DrawLine(start, end);

        float val = 0.0;
        char buffer[100];

        // Draw start label
        sprintf(buffer, "%.2f", val);
        DrawText(buffer, left, bottom+hunit, 10);

        // Draw the six bars, separating the five section
        for (int i = 0; i <= 5; i++) {
                start.x = left;
                start.y = bottom - hunit;
```

```
                end.x = left;
                end.y = bottom + hunit;

                DrawLine(start, end);

                left += unit;
                val  += unit;
        }

        // Draw end label
        sprintf(buffer, "%.1f", val - unit);
        DrawText(buffer, left - unit, bottom+hunit, 10);
}
```
