On Reductions in Dynamic Degrees of Freedom

Candidate: 52804

1 Introduction

'Coordinated movement is taken to be not just peripheral output, but an active ingredient in perceptual and cognitive processes, both in development and in adult function' [14].

A newborn infant has very little control over its movements and must acquire the motor skills required to enable exploration of its environment. Neo-natal infants show many spontaneous motions often ascribed as reflexes. Converging evidence has emerged however that many of these movements are prospective goal directed activity, that is action 'initiated by a motivated subject, defined by a goal and guided by information' [18]. This is not to say goal directed-behaviour is the foundation of all action, Piaget's primary circular reactions are one example of behaviour that does not appear to have an explicit goal [10].

The generation of exploratory motion can aid the formation of a basic movement repertoire. Exploration allows the infant to sense the effect of various movements in the environment, which guided by sensory-motor feedback, helps to shape these movements and the selection of movements into actions [14] [16]. Several factors shape the creation and refinement of these actions. First, the relationship between the infant and the environment forms a dynamic system where the results of actions modify the environment and the environment modifies the selection of actions. Second, the continuing development of the central nervous system, which undergoes massive changes in connectivity over the first few months of life, affects the selection of behaviours. Finally, the changes in musculoskeletal apparatus vary the dynamics associated with actions.

In executing any movement an infant is faced with significant challenges. The complexity of developing motor systems and the continuing change in the neural circuitry make acquiring and refining movement problematic. To execute a movement there must be some mechanism to compute the desired values for each joint, muscle and limb and create temporary functional assemblies of motor neurons and muscles into actions that can move the limbs under varying external loads.

One particularly important skill for infants develop is directed reaching, which enables an infant to explore its environment and bring interesting objects towards it for investigation. Experimental evidence has shown that neo-natal infants will prospectively reach towards objects they are fixated on [18].

von Hofsten [17] tracked the hand position of infants during the onset of reaching actions. He reported that for very young infants, the hand is constantly accelerating and decelerating, which results in initial movements that are very jerky. After 31 weeks infants' reaching behaviours begins to match adults where there is a single acceleration and deceleration of the hand during a reach.

Much of the developmental process for acquiring, refining and selecting well-coordinated and precise reaching movements is unknown. Many researches believe that infants use an exploratory search to assemble these movements [4] [12] [18]. However, the arm is a complex dynamic system that is continually changing in terms of its mechanical properties, up to maturation. In addition actions are often experienced under different external loads. The joints offer multiple degrees of freedom and for every reach, there are a immeasurable number of possibilities to move the limb through space to arrive at the required point.

If exploratory searching is used, then the search space is vast, which renders the likelihood of a successful result unlikely. The number of degrees of freedom is far greater than dimensions of the space through which limbs move, resulting in redundancy in motor signals. While a reach can be defined as still the same action regardless of how its executed because the goal is consistent, many different motor signals can lead to the same outcome. Conversely similar motor signals can lead to different outcomes under non-identical initial conditions.

Bernstein defined motor co-ordination as ‘the process of mastering redundant degrees of freedom of the moving organ, in other words its conversion to a controllable system’ [2] and proposed that control, and thus exploration, is constrained by reducing the degrees of freedom. Reduction occurs through the formation of synergies which act as a single unit of control. Synergies can be created by anatomical constraints, which reduce the kinematic degrees of freedom, or through the assemblage of functional units, which reduce the dynamic degrees of freedom. By constraining the degrees of freedom the complexity of the search problem is vastly reduced, which simplifies the process of learning to reach. With constraints reducing the scope of the task, only a limited number of muscles need to be driven by motor signals. Similar motor neuron patterns generate similar movements and the evaluation of those movements becomes simpler. Once a simple reaching movement has been mastered then it can be modified through experience to generate refined movement patterns.

1.1 Biological Evidence for Synergies in Reaching Tasks

Synergies have been observed in many biological and human studies and much work has been down on understanding the dynamics of reaching tasks [4] [5] [6] [8].

In Gottlieb et al [8] human subjects were asked to point to a series of targets. Experiments were varied for speed of pointing task and for external load by attaching weights to the wrists. For two degrees of freedom their experiments showed that the impulses produced at the shoulder was proportional to that produced at the elbow under varying loads and speeds. The torques at the two joints were closely synchronised with both approaching their maximum and minimum values almost simultaneously. Synergy arises because of the ‘... planing and generation of synchronised biphasic muscle torque pulses that remain near linear proportionality to each other throughout most of the movement.’ [8].

Berthier et al [4] reported that with the onset of reaching, young infants maintain a constant distance between the shoulder and the hand. This reduces the reaching problem to one similar to the manipulation of a simple spherical pendulum, reducing reaching to only two degrees of freedom. A secondary benefit is, that during the refinement of this reaching behaviour, the synergy ‘locks’ the elbow and wrist joints. Locking reduces the intersegmental forces on the joints, ensuring that the developing joints are less susceptible to damage.

1.2 Evolutionary Robotics

The Bernstein problem has also been explored in a number of epigenetic robotics studies [3] [7] [11] which have not presupposed the existence of synergies but tried to examine how synergies may be assembled and to quantify their impact on performance.

While reducing the dynamic degrees of freedom has been shown to improve searching performance, Berthouze and Lungarella [3] showed that releasing constrained degrees of freedom results in instability when varying the task load for a bouncing robot. Instead, in a highly interesting result they showed that periods of reduction and release need to be alternated to yield stable behaviour.

In Rohde and Di Paolo [11] a simulated robot arm was evolved to point at a series of targets. While linear synergies were not observed during unconstrained evolutionary searching, forcing linear synergies between joints improved searching performance. In addition, coupling controllers together showed little evolutionary advantage suggesting that ‘... the mere possibility of implementing constraints between effectors in a network does not provide a selection advantage’. The most interesting result in this paper is that adding degrees of freedom to the model improved its performance, because it allowed the simulated arm to exploit the dynamics of its environment.

In a recent study by Gomez et al [7] a simulated robot had to move an object into the centre of its field of vision. By constraining the system in a series of developmental stages, though increasing sensor precision, freeing and freezing of dynamic degrees of freedom and adding neurons, learning performance for the task was improved by 50%.

1.2.1 Hypothesis

Few epigenetic robotic studies have looked at how synergies shape bi-manual coordination [15]. For such movements, the potential number of degrees of freedom has increased considerably over a uni-manual reaching task and thus there is added impetus to constrain the complexity of learning and executing the required actions. It has been shown experimentally that for some bimanual reaching tasks, bi coordinated movement took longer than uni-manual reaches, reflecting the added complexity introduced by bi-manual co-ordination [6].

While synergies provide a useful explanation for how to solve the complex movement issues required for limb control, there are many other open questions. How are synergies selected for? How do beneficial synergies arise? How are they organised, if they are seen to appear and reappear, ?

In this paper, a series of minimal evolutionary robotics models are used to investigate the emergence of synergies and their affects on co-ordinated movement.

The hypothesis of this paper is that in unconstrained evolutionary searching, we should expect synergies to arise given the requirements for complexity reduction in bi-manual reaching. In addition, given biologically identified constraints, synergies should improve the performance of evolution searches for effective solutions.

2 The Model

In the model under evaluation, a simulated agent must engage in a series of uni-manual and bi-manual reaching tasks. For each task the agent must touch a series of targets constrained on a fixed planar surface.

Each agent is modelled as a column with two hinged arms (See Figure 1). Sagittal symmetry is enforced in the agent with both arms having identical physical properties. Each arm is modelled as having an upper arm, lower arm and a small hand. In addition, each arm has a two simple hinge joints given 1 degree-of-freedom (DOF) at the shoulder and 1 DOF at the elbow. There is no joint between the hand and the lower arm. The joints are constrained to move in a manner approximating human physiology; the shoulder joint can move approximately 225^{O} and the elbow $\pm 90^{\mathrm{O}}$ on the fixed plane. The arm is constrained to move only in a fixed horizontal plane, to simplify the model, limiting motion to two degrees of freedom.

Each arm has its own separate controller, which moves the arm by applying a torque value (t) at each joint.

The agents are evolved to solve a uni-manual and a bi-manual reaching task. In each task, two controllers are compared. The first controller is used as a baseline for comparison and is referred to as an unconstrained controller . The second type of controller has a forced synergy between the torque of the shoulder and elbow joint. The relationship between the shoulder and elbow joints is constrained to be a simple linear relation where $t_s = Kt_e$ [8] and K is in the range $\in 2.5 - 3.0$.

In the uni-manual reaching task, a single limb of an agent is manipulated. This has strong parallels with Rhode and Di Paolo's work [11]. The unconstrained controller is identified as UCU and the forced synergy controller by FSB.

The bi-manual reaching task is governed by the same criteria as the uni-manual task but the movements of both arms are simulated. Each arm has its own independent control system. As in in the previous section,

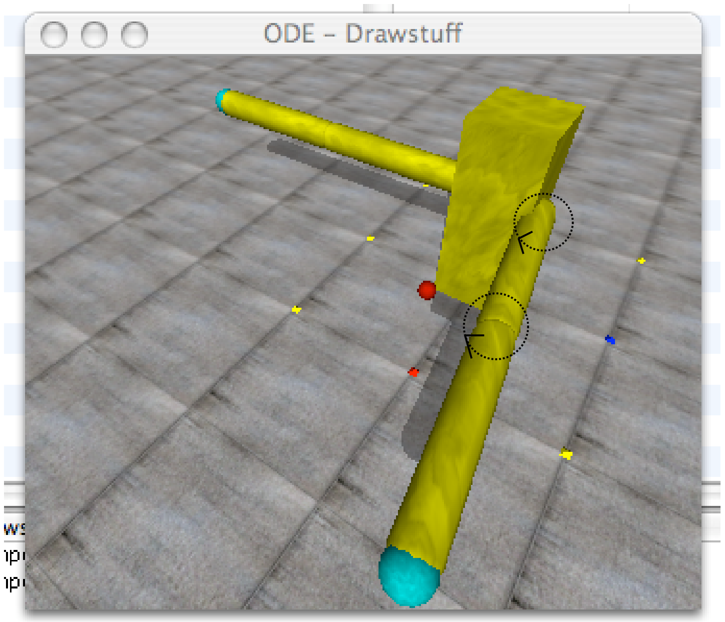


Figure 1: **Rendering of agent in ODE.** Each limb has 2 DOF, one in the shoulder and one in the arm. The robot's task is to touch the target point (shown as a red sphere), with both 'hands'.

the dynamics of the system are first examined when there is no coupling between the controllers (UCB). The task is then studied where synergies have been introduced between the controllers (FSB).

The model is written in C++ and uses the Open Dynamics Engine (ODE) [13] to simulate the agent.

2.1 The Controller

Each arm is controlled by a fully connected Continuous Time Recurrent Neural Network (CTRNN) [1], where each neuron in the network is described by equation 1.

$$\tau_i \dot{y}_i = -y_i + \sum_{j=1}^{N} w_{ji}\sigma(y_j + \theta_j) + I_i \tag{1}$$

y_i is the cell's potential, τ_i is the decay constant (range $[0.01, 4]$), w_{ji} is the strength of the connection from the j^{th} to the i^{th} neuron (range $[-8, 8]$), θ_j is the node's bias (range $[-3, 3]$) and I_i represents the current external input to this node, where i varies between 1 and N. N is the number of nodes in this network.

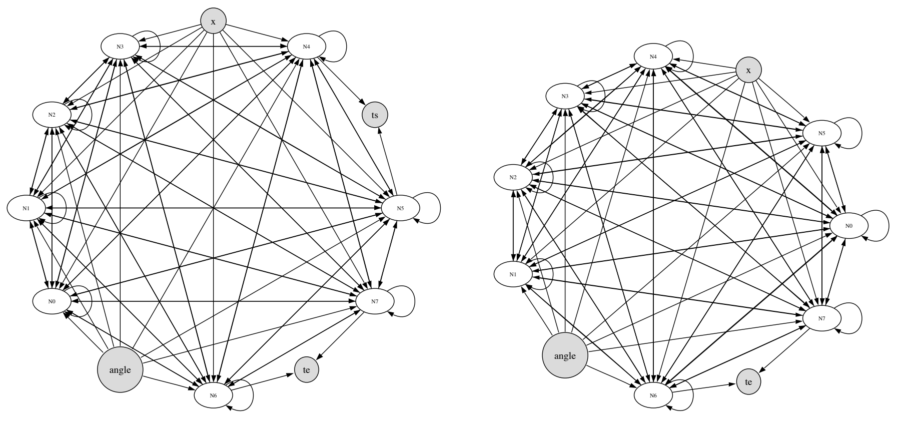
σ is the standard logistic activation function common to many neural networks, scaling its output $\in [0, 1]$ and is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

The CTRNNs are formed into simple minimal controllers, as shown in Figure 2. The current distance of the hand to the target is provided as two inputs (representing x and y components) along with the angle of the target. These inputs are mapped, via an evolvable layer of input gains ($\in [-5, 5]$), to each neuron in the network. The joint's torque is generated using (3), where $m1$ and $m2$ are a pair of antagonist motor neurons whose activations are scaled by an evolvable motor gain, M ($\in [0.1, 10]$).

$$\Delta t = M.(m1 - m2) \tag{3}$$

Antagonistic motor neurons are used to stabilise joint movement and aid searching by the genetic algorithm.



(a) Unconstrained controller (b) Constrained controller

Figure 2: **CTRNN Controller for one arm**. Distance from source (in the x plane) and target angle are provided as input to all nodes via an evolvable layer of weights. In the constrained controller torque for the shoulder joint is calculated via a linear scaling of the elbow joint.

2.2 Genetic Algorithm

A population of agent controllers is evolved using a Microbial tournament Genetic Algorithm [9]. At every iteration of the genetic algorithm (GA), two controllers are selected at random to compete against each other. Each controller is assessed against a series of point targets. The fitness scores of both agents are assessed and the controllers are bred, using crossover and mutation, to form a new offspring, which replaces the lowest scoring controller.

All the components of the genotype are constrained to be in the range $\in [0, 1]$ and are linearly scaled to match the encoded parameter they represent. The exceptions are the input and decay time values which are exponentially scaled. Each loci is mutated with a probability of 0.5 and has a probability of recombination of 0.5. Any mutation that takes a value outside of its allowed range is truncated to conform to the set range.

At the beginning of each trial, the controllers are run without any target information for 50 iterative steps to allow the CTRNN to stabilise. The controller is then attached to the simulated robot and the series of trials begins.

The Microbial GA repeats until no perceivable fitness increase is observed.

2.3 Fitness Function

Each controller runs for a number of trials, typically 5. In the beginning of each trial the robot is set to a neural position (arms straight out to the sides) and then a target is presented at a random point within the reaching range of the arms. Each target is displayed for T steps where, T is drawn randomly from a range $[800 : 1400]$ time steps.

For a single robot arm, the fitness function is assessed as (4), where D_t is the distance between the nearest surface of the robot's hand to the target and D_0 is the starting distance. If $D_t > D_0$ then the contribution to fitness in this time step is 0. This function gives higher scores for controllers that reduce the distance between the robot's hand throughout the trial.

$$F_j(i) = \frac{1}{T}\sum_{j=1}^{n}(1 - \frac{D_t}{D_0}) \tag{4}$$

For the bi-manual reaching task the fitness of the ith individual on the jth target is given by (5), where fitness is calculated using both hand's positions. DL_t is the position of the left hand at timestep t and DR_t is the position of the right hand at this time step.

$$F_j(i) = \frac{1}{2T}\sum_{j=1}^{n}(1 - \frac{DL_t}{DL_(0)}) + (1 - \frac{DR_(t)}{DR_(0)}) \tag{5}$$

The overall fitness is given by averaging the fitness on each trial, minus 20% of the standard deviation to prefer solutions with low variability.

3 Results

The following results are provisional as simulation errors prevented the evolution of successful and stable reaching behaviours.

3.1 Unconstrained unimanual reaching - UCU

Figure 3 shows the worst, best and average fitness scores for the population during evolution for the unconstrained uni-manual reaching task. The fitness profile is typical of runs seen under these conditions with little appreciable increase in fitness.

At the evolution the behaviour of the fitness member of the population is then examined for another series of targets. Figure 4 shows the distance of the hand from the target over this run as well as the the input and output values for the controller. This agent exhibits typical behaviour by moving its arm rapidly through the space near each target.

Finally, Figure 5 shows the relationship of shoulder and elbow torques. For unconstrained agents there often appears to be a non-linear relationship between the torques, though not in this case. However, this relationship seems to do more to with the insensitivity of the controller to its inputs and the torque values approaching constant values over the lifetime of simulation.

3.2 Constrained unimanual reaching - FSU

With a forced synergy between shoulder and elbow torques, little difference in evolution fitness or task performance. Figures 6 and 7 and **??** show the evolutionary fitness, stability of behaviour and torque plots from one evolutionary run. This agent held its arm in a constant position for most of the run, exploiting the fact that the targets were often not far from the centre of the agent's body.

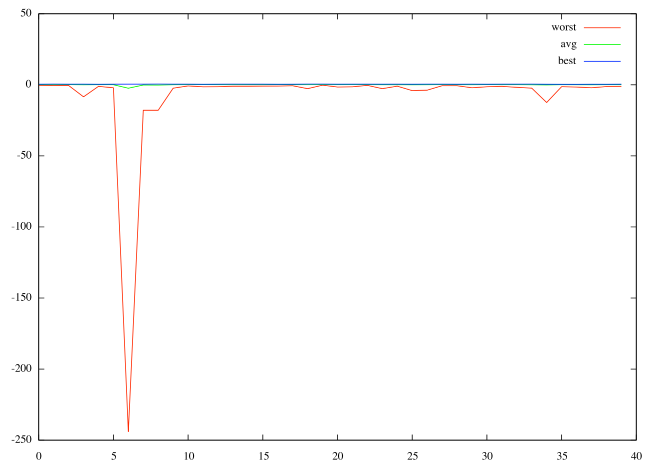
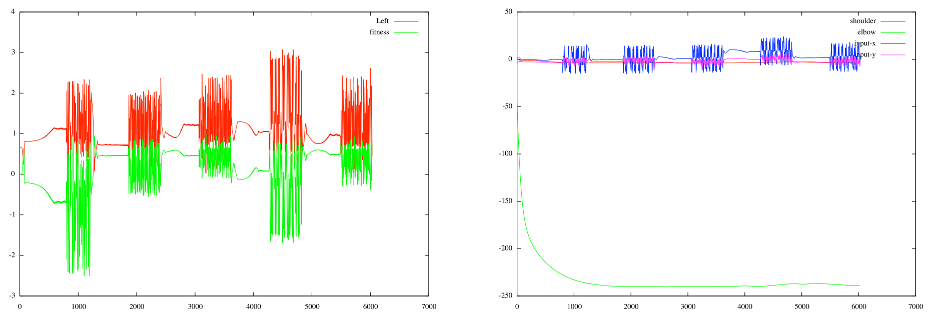


Figure 3: **Fitness of population during evolution per epoch**. Each epoch consists of 40 tournaments.



(a) Distance to source (b) Internal inputs and outputs

Figure 4: **Stability of behaviour** for a series of target presentations and corresponding controller input and output.

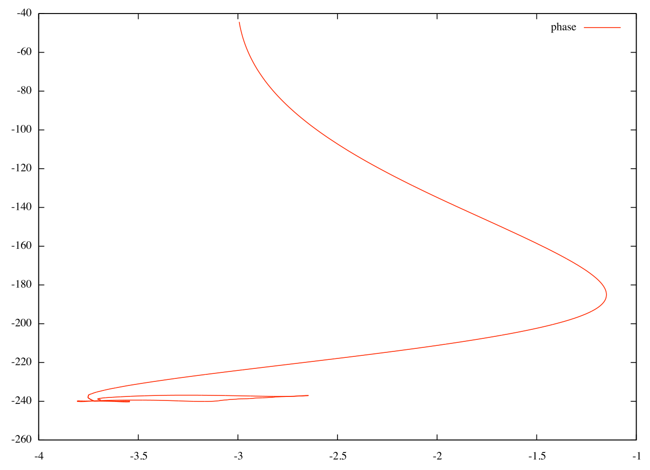


Figure 5: **Shoulder versus elbow torque.**

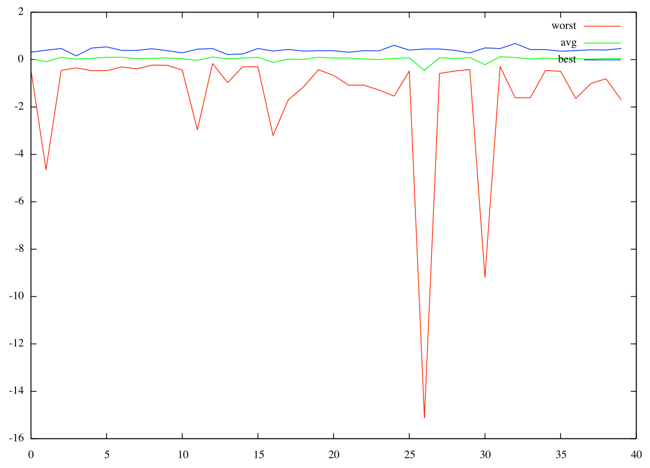
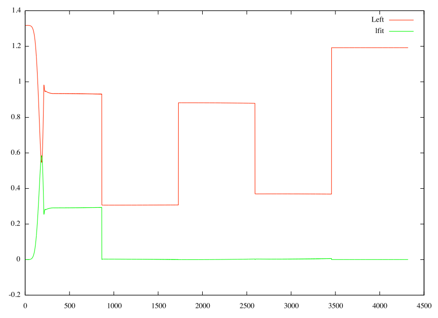
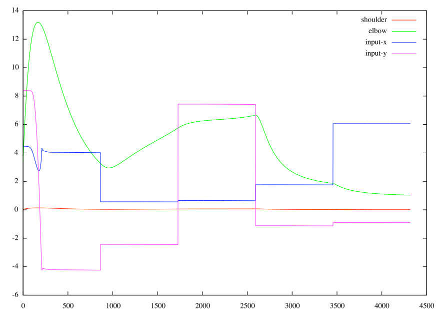


Figure 6: **Fitness of population during evolution per epoch - FSU**. Each epoch consists of 40 tournaments.



(a) Distance to source



(b) Internal inputs and outputs

Figure 7: **Stability of behaviour - FSU**. Distance to target and corresponding controller input and output.

3.2.1 Further functional synergies

Berthier et al [4] observed half of the infants under study started their reach from the similar starting positions each time, resulting in a further functional synergy by creating a common starting point. In this simulation the arm is repositioned at the starting position (arm straight out to the side) when a new target appears. However, due to simulation issues, no noticeable difference in controllers was observed through the introduction of this functional synergy.

4 Discussions and Conclusions

Rohde and Di Paolo [11] concluded by stating that '...improving evolvability is not a matter of scaling up or down the search space but of *reshaping the fitness landscape*'. Unfortunately, the fitness landscape offered by this simulation prevented the evolution of solutions that were observed to be stable. Regardless of parameter tuning, the evolutionary search quickly plateaued resulting in little performance in controllers. Figure 8 shows a typical plot for the worst, average and best fitness seen in every 30 steps of the Microbial GA. Evolution exploited the dynamics of the simulated arm to move very quickly and rapidly in a space near but not necessarily around the target. Due to time constraints, the reason for this behaviour has not been ascertained.

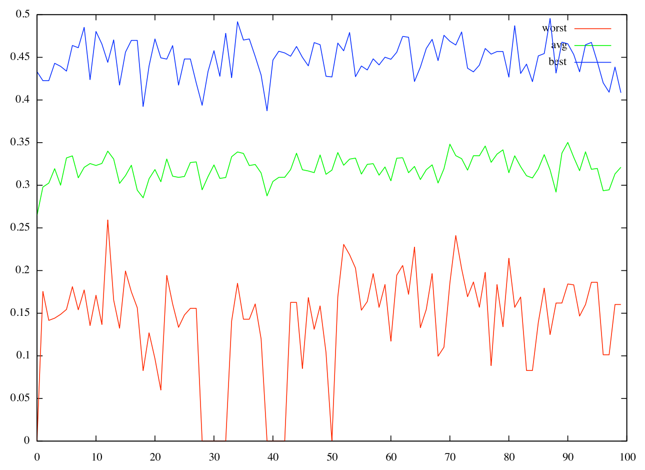


Figure 8: **Typical fitness results during evolution.** These figures were taken from an UCU run but were typical of most observed evolutionary runs. Each x-axis point represents 30 tournaments in the GA.

However, in other studies [7] [11] the introduction of synergies improved performance. Synergies can be viewed as a way of structuring information [15]. Early developmental stages have limited cognitive and musculoskeletal control. Therefore by reducing the degrees of freedom there is also a reduction in the amount of information and thus the complexity of learning actions is diminished.

This fits into the view that development is an incremental process that moves in stages from one development system to the next, with each stage becoming more complex [16]. Berthier [4] observed that the constraint on hand and shoulder positions, which reduces reaching to a two degrees of freedom problem, is probably enforced by physiology immaturity due to proximodistal direction of maturation. Tracked hand speed is consistent across the reported data suggesting a synergy that constrains the sub-space allowing generalisation of simple reaching. As the musculoskeletal and central nervous systems mature then the release of constraints occurs.

In a recent paper, d'Avella et al [5] showed how the central nervous system may organise synergies to create the appropriate muscle activation patterns for actions. In a fast-reaching task, they identified a small number of synergies that could be used to reconstruct the muscle patterns accurately. This work highlights how combinations of synergies reduce the dimensionality of the control problem and thus the information associated with it, to a one of a much lower dimensionality. However, Berthouze and Lungarella [3] showed that alternating periods of reduction and release were required to have stable behaviour showing that synergies alone may not be sufficient to reduce complexity.

Much of how the central nervous system creates and selects appropriate muscle patterns is still unknown. Therefore further work is required to understand how synergies can be combined and organised to shape the required muscle patterns.

While the results of the simulation were disappointing, physical simulations like this experiment have much to offer. While it is technically challenging to have a physical robot that undergoes musculoskeletal development, simulated robots can easily have their morphology, controllers and sensory motor loops altered. Great care must be taken to ensure that simulated systems reflect the behaviour of actual developmental systems in a meaningful way for such simulation to be of use.

References

[1] Beer, R.D. (1995) *On the dynamics of small continuous-time recurrent neural networks.* Adaptive Behavior, Vol. 3(4), pp. 469-509.

[2] Bernstein, N.A. (1967) *The co-ordination and regulation of movements.* Pergamon Press, Oxford.

[3] Berthouze, L and Lungarella, M. (2004) *Motor skill acquisition under environmental perturbations: On the necessity of alternate freezing and freeing of degrees of freedom.* Adaptive Behavior Vol. 12, pp. 47-64.

[4] Berthier, N.E., Clifton, R.K., McCall, D.D. and Robin, D.J. (1999) *Proximodistal structure of early reaching in human infants.* Experimental Brain Research, Vol. 127, pp. 259-269.

[5] d'Advella, A., Portone, A., Fernandez, L. and Lacquanti, F. (2006) *Control of fast-reaching movements by muscle synergy combinations.* Journal of Neuroscience, Vol. 26(30), pp. 7791-7810.

[6] Garry, M.I. and Franks, I.M. (2000) *Reaction time differences in spatially constrained bilateral and unilateral movements.* Experimental Brain Research, Vol. 131, pp. 236-243.

[7] Gómez, G., Lungarella, M., Eggenberger Hotz, P., Matsushita, K. and Pfeifer, R. (2004) *Simulating development in a real robot: on the concurrent increase of sensory, motor, and neural complexity.* Proc. 4th International Workshop on Epigenetic Robotics, pp. 119-122

[8] Gottlieb, G.L., Song, Q., Hong, D.-A., and Corcos, D.M. (1996) *Coordinating two degrees of freedom during human arm movement: Load and speed invariance of relative joint torques.* Journal of Neurophysiology, Vol. 76, pp. 3196-3206.

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

[10] Piaget, J. (1952) *The origins of intelligence in children.*International Universities Press.

[11] Rohde, M. and Di Paolo, E. (2005) *t for two: Linear synergy advances in the evolution of directional pointing behaviours.* Proc. of ECAL 2005, LNAI 3630, pp. 262-271.

[12] Sheya, A. and Smith, L.B. (2006) *Development through sensory-motor coordinations.*, In press.

[13] Smith, R. (2006) *Open dynamics engine.* http://www.ode.org

[14] Sporns, O. and Edelman, G.M. (1993) *Solving Bernstein's problem: A proposal for the development of coordinated movement by selection.* Child Development, Vol. 64, pp. 960-981.

[15] Sporns, O. and Lungarella, M. 2006. *Evolving coordinated behavior by maximizing information structure.* In Rocha, L. et al. eds. Artificial Life X. Cambridge, MA: MIT Press.

[16] Thelen, E. and Smith, L.B. (1994) *A dynamic systems approach to the development of cognition and action.* MIT Press.

[17] von Hofsten, C. (1991) *The structure of early reaching movements: A longitudinal study.* Journal of Motor Behavior , Vol. 23(4), pp. 280-292.

[18] von Hofsten, C. (2004) *An action perspective on motor development.* Trends in Cognitive Science, Vol. 8(6), pp. 266-272.

[19] Whiting, H.T.A. (editor) (1984) *Human motor actions : Bernstein reassessed.* North Holland.

[20] Zaal, F., Daigle, K., Gottlieb, G.L. and Thelen, E. (1999) *An unlearned principle for controlling natural movements.* Journal of Neurophysiology, Vol. 82, pp. 255-259.

Appendix A - Code

AgentModel.h

```
/*
 *  AgentModel.h
 *  DOD
 */

#pragma once

#include "SimulationRecord.h"

/**
 *
 */
#define kWINDOW_HEIGHT  600

#define kWINDOW_WIDTH   600
/**
 *  maximum number of geometries per body
 */
#define kMAX_VISUALS 10

/**
 * Defines the height of the target off the ground
 */
#define kTARGET_HEIGHT  1.0


/**
 * Run the current model displayed in the simulation rec
 * If visualise is true - then show visual output (slow)
 * If visualise is false - then don't display (fast)
 */
void RunModel (SimulationRecord& rec, bool visualise);
```

AgentModel.cpp

```
/*************************************************************************
 *                                                                       *
 * Open Dynamics Engine, Copyright (C) 2001,2002 Russell L. Smith.       *
 * All rights reserved.  Email: russ@q12.org   Web: www.q12.org          *
 *                                                                       *
 * This library is free software; you can redistribute it and/or        *
 * modify it under the terms of EITHER:                                  *
 *   (1) The GNU Lesser General Public License as published by the Free  *
 *       Software Foundation; either version 2.1 of the License, or (at  *
 *       your option) any later version. The text of the GNU Lesser      *
 *       General Public License is included with this library in the     *
 *       file LICENSE.TXT.                                               *
 *   (2) The BSD-style license that is included with this library in     *
 *       the file LICENSE-BSD.TXT.                                       *
 *                                                                       *
 * This library is distributed in the hope that it will be useful,       *
 * but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the files    *
 * LICENSE.TXT and LICENSE-BSD.TXT for more details.                     *
 *                                                                       *
 *************************************************************************/

/*
```

```
  This code is taken from the 0.39 version of ODE
  Modified by Eric Vaughn and Greg Studer.

  Modified by Pip
*/

#include <ode/ode.h>
#include <drawstuff/drawstuff.h>
#include <assert.h>
#include <stdlib.h>
#include <map>
#include <time.h>
#include <vector>
#include <string>

#include "Globals.h"
#include "Genetics.h"
#include "Simulation.h"
#include "SimulationRecord.h"
#include "AgentModel.h"
#include "Utils.h"

#ifdef MSVC
#pragma warning(disable:4244 4305)  // for VC++, no precision loss complaints
#endif

//----- Local Pre-proc definitions ----------------------------------------------------

// select correct drawing functions
#ifdef dDOUBLE
#define dsDrawBox                               dsDrawBoxD
#define dsDrawSphere                    dsDrawSphereD
#define dsDrawCylinder                  dsDrawCylinderD
#define dsDrawCappedCylinder    dsDrawCappedCylinderD
#define dsDrawLine                              dsDrawLineD
#endif

// Max torque allowed
#define kMAX_TORQUE                             500

// Define ODE_DIRECTORY macro
#define QUOTE(_text) #_text
#define ODE_DIRECTORY(_dir) QUOTE(../../ode-0.7/##_dir)

// some constants
#define MAX_OBJECTS 30      // max number of objects
#define MAX_CONTACTS 4          // maximum number of contact points per body

/*
 * Stores information about a specific geom. This is used by the simulation
 * loop to draw geoms.
 */
struct Visual {
  dGeomID geom;
  dReal red;
  dReal green;
  dReal blue;
  dReal alpha;
  int drawCap;   // draw the cap of a cylinder. Remember: This does not affect collisions they will still hit the cap.
  int highlight; // highlight this Object for debugging
};

/*
 * Stores information about objects in ODE. Essentially this just provides
 * a way for the simulation loop to draw the geoms connected to the body.
 */
struct Object {
  dBodyID       body;                                  // the body
  Visual        visuals[kMAX_VISUALS];  // visuals holding geom color and id
  int           numOfVisuals;
};


/**
 * Stores information about the current target object
 */
struct Target {
        dGeomID geom;                   //!< Targets geom
        dReal   x;                              //!< Targets x position
        dReal   y;                              //!< Targets y position
        float   ang;                    //!< Angle of target (in degrees)
        float   arm_length;
};

struct Hand {
        dBodyID body;
        dGeomID g;
        dReal   x;
        dReal   y;
        dReal   z;
};

struct Torques {
        dReal t1;
        dReal t2;
        dReal t3;
        dReal t4;
};

//---- Globals -------------------------------------------------------------------

static dWorldID world;                                          // the ode simulation world
static dSpaceID space;                                          // the root ode simulation space

static Object gObjects[MAX_OBJECTS];            // objects in the simulation
static int gNumOfObjects=0;                                     // number of objects in simulation

static dJointGroupID contactgroup;                      // joint group used for storing / deleting
                                                                                // contact joints
static bool c_friction    = true;                       // toggle coulomb friction

/*
 * Information related to controlling joints
 */
typedef void (*ControllerCallback) (float simTime, dJointID jid);
struct JointInfo
{
        dJointID                        joint;
        ControllerCallback      proc;
};

typedef std::map<int, JointInfo> Joints;
Joints* gJoints = 0;

/**
 * Count of the number of joints crated
 * Used to allocate joint ids for easy indentification
 */
static int gJointCount = 0;
static float    gUpdateTime = 0;

/**
 * Time to update joints
 * TODO: need to find a better way of applying Torque
 */
static float    gUpdateInterval = kTIME_STEP * 5.0;

/*
 * Current hand objects
 */
Hand lhand = {0, 0, 0.0, 0.0, 0.0};

#if kBI_MANUAL
Hand rhand = {0, 0, 0.0, 0.0, 0.0};
#endif

/**
 * Holds the geomID for the target
 */
static Target gTarget = {0, 0.0, 0.0, 0.0, 0.0};

static dGeomID gBody = 0;
static const float   b_x = 0.2;
static const float   b_y = 0.2;
static const float   b_z = kTARGET_HEIGHT + 0.5;

/**
 * Holds the target radius
 */
static float gTargetRadius      = 0.05;
static float gTargetZ                   = kTARGET_HEIGHT;

/**
 * If true then contact has occurred between a hand and target
 */
static bool  gContact                   = false;

/**
 * Record current joint torques
 */
static Torques gTorques;

static SimulationRecord* gRec   = 0;

/**
 * If false then we haven't run drawstuff before so display options etc.
 */
static bool      gInit                          = false;

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

static bool BuildRobot ();

// Utility helper functions
static Object*  NewObject               ();
static Visual*  AddGeom                 (Object* obj, dGeomID geom, dReal red, dReal green, dReal blue);
static void             CreateWorld             ();
static void             DestroyWorld    ();
static void             DrawGeom                (dGeomID g, const dReal *pos, const dReal *R, int show_aabb, int cap);

// Drawstuff callback functions
static void                     Stop            ();
static void                     Start           ();
static void                     SimLoop         (int pause);
static void                     Command         (int cmd);
static void                     mySimLoop   ();

// ODE Collision callback function
static void             NearCallback(void *data, dGeomID o1, dGeomID o2);

// Shared code to get hand positions
void GetHandPositions (float& lx, float& ly, float & rx, float& ry);
static void UpdateJoints ();

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

// NOTE: Box geoms are preferred to spheres as there seems to be a rendering bug
// during spheres where they ‘disappear’ from view. Shadows are given but ODE doesn’t
// seem to draw sphere geoms when restarted

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

int main (int argc, char **argv)
{
        // Seed random number generation
        srand(static_cast<int>(time(NULL)));

        if (argc == 3) {
                if (strcmp (argv[1], "-f") == 0) {
                        std::string fileName = argv[2];
                        Eval(fileName);
                }
        }
        else {
                EvalFitest();
                // Call evolutionary algorithm
                printf("\nEvolving......\n");
                Evol();
                EvalFitest();
        }
        return 0;
}

//------------------------------------------------------------------------------
// Creates a reaching target for the simulation.
// Attached as a geometry without a body and is handled as a special case in the
// collision dectection code.
void GenerateTarget (SimulationRecord & rec)
{
        rec.ClearTemporary();

        // Create Target
        if (gTarget.geom != 0) {
                dGeomDestroy(gTarget.geom);
        }

        // Randomise target - in a reaching circle around the agent
        // Angle of object needs to be between 0-90 and 270-360 to be in the quarters in
        // front of the robot due to the orientation of ODE’s projection.
        float angle  = drand48();                       // 0.1

        // Can re-write range as 270-450 degrees
        float adjusted = RescaleUnitValue(270.0, 450.0, angle);

        // Adjust and normalise angle so its between 0-360 degrees
        gTarget.ang  = NormaliseAngle(adjusted);

        // Place target between 1/4 and 3/4’s of the length of the robot arm
        float min_len = gTarget.arm_length/4.0;
        gTarget.x = static_cast<float>(RangedDouble(min_len, min_len * 3.0));
        gTarget.y = 0.0;

        // Rotate around origin
        RotatePointAroundPoint(gTarget.x, gTarget.y, 0.0, 0.0, gTarget.ang);

        // Generate new target pos
        gTarget.geom = dCreateBox (space, gTargetRadius, gTargetRadius, gTargetRadius);
        dGeomSetPosition(gTarget.geom, gTarget.x, gTarget.y, kTARGET_HEIGHT);

        // Fill out starting info
        float lx, ly, rx, ry;
        GetHandPositions(lx, ly, rx, ry);

        // Don't care about sign of distance
        rec.start_ldistance = fabs(DistanceBetweenPoints(lx, ly, gTarget.x, gTarget.y));
        rec.tangle          = gTarget.ang;

#if kBI_MANUAL
        rec.start_rdistance = fabs(DistanceBetweenPoints(rx, ry, gTarget.x, gTarget.y));
#endif

        rec.steps = 0.0;
        rec.targets++;
}

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

void RunModel (SimulationRecord& rec, bool visualise)
{
        // setup pointers to drawstuff callback functions
        dsFunctions fn;
        fn.version                      = DS_VERSION;
        fn.start                        = &Start;
        fn.step                         = &SimLoop;
        fn.command                      = &Command;
        fn.stop                         = &Stop;

#if MAC_ENV
        fn.path_to_textures     = "/Users/pip/Documents/Sussex/DOD/Project/DOD/ode-0.8/drawstuff/textures";
#else
        fn.path_to_textures     = ODE_DIRECTORY(drawstuff/textures);
#endif

        // Record as current record
        gRec = &rec;

        // create world
#if !WIN_ENV
        dInitODE();
#endif
        CreateWorld();

        gTorques.t1 = gTorques.t2 = gTorques.t3 = gTorques.t4 = 0.0;

        // Generate 'robot'
        BuildRobot();

        // Generate initial target
        GenerateTarget(rec);

        // run simulation
        if (visualise) {
                dsSimulationLoop (0, NULL, kWINDOW_WIDTH, kWINDOW_HEIGHT, &fn);
        }
        else {
                mySimLoop();
        }
        DestroyWorld();
        dCloseODE();
}

//-------------------------------------------------------------------------------
//  SIM START CALLBACK FUNCTION
//
// Used to initialize the drawstuff simulation.
static void Start()
{

        // run simulation
        if (gInit == false) {
                dsPrint("\nDOD project options:");
                dsPrint("\n\tPress '1' for front view");
                dsPrint("\n\tPress '2' for side view");
                dsPrint("\n\tPress '3' for top view");
                dsPrint("\n\tPrint 'v' to get current viewpoint");
                dsPrint("\n\tPress 'q' to quit the simulation loop.\n\n");

                gInit = true;
        }

        // NOTE: need to set view as default view is unusable
        // when running if the 'v' key is pressed it will give you a new xyz and hpr just past them here
        // to change the starting view location.
        static float xyz[3] = {2.898957f, -0.017567f, 1.630001f};
        static float hpr[3] = {-177.000000f, -26.500000f, 0.000000f};

        dsSetViewpoint (xyz,hpr);
}

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

void Stop ()
{
        ;       // Tidy up
}

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

void GetHandPositions (float& lx, float& ly, float & rx, float& ry)
{
        lx = ly = rx = ry = 0.0;

        if (lhand.g) {
                const dReal *pos = dGeomGetPosition(lhand.g);
                lx = pos[0];
                ly = pos[1];
        }

#if kBI_MANUAL
        if (rhand.g) {
                const dReal *pos = dGeomGetPosition(rhand.g);

                rx = pos[0];
                ry = pos[1];
        }
#endif
}
```

```
//------------------------------------------------------------------------------
// Work out input values
bool UpdateTorques()
{
        bool cont = true;

        // First check we have no more targets to generate
        if (gRec->targets > gRec->max_targets) {
                cont = false;
        }
        else if (gRec->steps > gRec->max_steps) {

                // Cache fitess values for trial
                CalcFitness(gRec);

                // power down joints
                gRec->lstorque = gRec->letorque = 0.0;
                gTorques.t1 = gTorques.t2 = gTorques.t3 = gTorques.t4 = 0.0;

#if kBI_MANUAL
                gRec->retorque = 0.00;
                gRec->rstorque = 0.00;
#endif
                // Reset joint torques
                UpdateJoints();

                // Reset position of hands
#if kRESET_HANDS
                dBodySetPosition(lhand.body, lhand.x, lhand.y, lhand.z);
#if kBI_MANUAL
                dBodySetPosition(rhand.body, rhand.x, rhand.y, rhand.z);
#endif
#endif
                GenerateTarget(*gRec);
        }
        else {
                float x1, y1, x2, y2;
                GetHandPositions(x1, y1, x2, y2);
                cont = IntegrateSingleStep(gRec, x1, y1, x2, y2, gTarget.x, gTarget.y);
        }
        return cont;
}

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

void UpdateJoints ()
{
        Joints::iterator pos = gJoints->begin();
        // If we're controlling anything, do that first
        for (; pos != gJoints->end(); ++pos) {
                JointInfo info = pos->second;

                if (info.proc) {
                        (*info.proc)(gRec->step_size, info.joint);
                }
        }
}

//------------------------------------------------------------------------------
// MAIN SIMULATION LOOP CALLBACK
//
// Handles the updating of the world for the next time step,
// called by drawstuff when it's ready for a change.
static void SimLoop (int pause)
{
```

```
double dt        = dsElapsedTime();
double steps     = dt/ kTIME_STEP;
int    num_steps = (int) ceilf(steps);
bool   ok    = true;


// Update the simulation if not paused
 for (int i=0; (i < num_steps) && (!pause) && (ok); i++) {

        // Collide all the objects using our efficient space
        dSpaceCollide(space, 0, &NearCallback);

        // Update the CTRNN
        ok = UpdateTorques();

        // Update the joints
        gUpdateTime += gRec->step_size;

        if(gUpdateTime >= gUpdateInterval) {
                gUpdateTime -= gUpdateInterval;

                UpdateJoints();
        }

        // Update the world
        dWorldStep(world, gRec->step_size);

        // Remove all contact joints
        dJointGroupEmpty (contactgroup);
}

//---- Visualise Geoms
if (ok) {
        // Set a default color + texture for drawing
        dsSetColor(1, 1, 0);
        dsSetTexture (DS_WOOD);

        // Draw all the GEOMS in the new positions
        for(int i=0; i< gNumOfObjects; i++) {
                for(int j=0; j < gObjects[i].numOfVisuals; j++) {
                        Visual* v = &gObjects[i].visuals[j];
                        dsSetColor (v->red, v->green, v->blue);
                        DrawGeom (v->geom,0,0,v->highlight, v->drawCap);
                }
        }

        // Draw target
        // Is not attached to a body so has to be handled separately
        if (gTarget.geom != 0) {
                dReal pos[3] = {gTarget.x, gTarget.y, kTARGET_HEIGHT };
                const dReal *r = dGeomGetRotation (gTarget.geom);

                if (gContact) {
                        dsSetColor(0,0,1);              // blue
                }
                else {
                        dsSetColor (1,0,0);             // red
                }
                DrawGeom (gTarget.geom, pos, r, false, false);
        }

        if (gBody != 0) {
                dsSetColor(0,0,0);
                dReal pos[3] = {0.0, 0.0, b_z / 2.0 };
```

```
                        const dReal *r = dGeomGetRotation (gBody);
                        DrawGeom (gBody, pos, r, false, false);
                }
        }
        gContact = false;

        if (!ok) {
                dsStop();
        }
}


//-----------------------------------------------------------------------------
// Simulation without drawstuff
// TODO: joint update?
void mySimLoop ()
{

        bool ok = true;

         // Update the simulation if not paused
        while (ok) {
                // Collide all the objects using our efficient space
                dSpaceCollide(space, 0, &NearCallback);

                // Update the CTRNN
                ok = UpdateTorques();

                // Update the joints
                Joints::iterator pos = gJoints->begin();

                // If we're controlling anything, do that first
                // Update the joints
                gUpdateTime += gRec->step_size;

                if(gUpdateTime >= gUpdateInterval) {
                        gUpdateTime -= gUpdateInterval;

                        Joints::iterator pos = gJoints->begin();
                        // If we're controlling anything, do that first
                        for (; pos != gJoints->end(); ++pos) {
                                JointInfo info = pos->second;

                                if (info.proc) {
                                        (*info.proc)(gRec->step_size, info.joint);
                                }
                        }
                }

                // Update the world
                dWorldStep(world, gRec->step_size);

                // Remove all contact joints
                dJointGroupEmpty (contactgroup);

                gContact = false;
        }
}

//-----------------------------------------------------------------------------
// Resets the ODE world to a new one
void CreateWorld()
{
  // root world and space
        world = dWorldCreate();
        space = dHashSpaceCreate(0);

        contactgroup = dJointGroupCreate(0);
        dWorldSetGravity(world, 0, 0, -0.5);
        dWorldSetCFM(world, 1e-5);

        // Create an infinite plane to sit on
        dCreatePlane(space, 0, 0, 1, 0);

        // clear out our array of objects
        memset(gObjects, 0, sizeof(gObjects));

        gNumOfObjects                   = 0;
        if (gJoints == 0) {
                gJoints = new Joints;
        }
        else {
                gJoints->clear();
        }
        gJointCount = 0;
        gContact = false;

        gUpdateTime = 0.0;
        if (gRec) {
                gRec->lstorque = 0.0;
                gRec->letorque = 0.0;
        }
}

//----------------------------------------------------------------------------
// Cleanup the world
void DestroyWorld()

{
        dJointGroupDestroy (contactgroup);

        for (int i = 0; i < gNumOfObjects; i++) {
                dBodyDestroy(gObjects[i].body);
        }

        dSpaceDestroy (space);
        dWorldDestroy (world);


        // World automatically cleans up geoms
        gTarget.geom = 0;
        lhand.body = 0;
        lhand.g = 0;

#if kBI_MANUAL
        rhand.body   = 0;
        rhand.g      = 0;
#endif

        gBody   = 0;

        gJoints->clear();
}

//----------------------------------------------------------------------------
// Get a new object. For performance (and simplicity), 'new' or 'malloc' are
// not used. Instead we just return a pointer to a statically allocated Object.
static Object* NewObject()
{
  assert(gNumOfObjects < MAX_OBJECTS);
  // return a new object from our array
  Object* obj = &gObjects[gNumOfObjects];
  gNumOfObjects++;

  // initialize instance variables
  obj->body = 0;
  obj->numOfVisuals = 0;

  return obj;
}

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

dJointID NewJoint (ControllerCallback proc)
{
        JointInfo info;
        info.proc = proc;

        info.joint = dJointCreateHinge(world, 0);

        gJoints->insert(std::pair<int, JointInfo> (gJointCount++, info));
        return info.joint;
}

//---------------------------------------------------------------------------
// Add the geom to Object structure with its colour.
static Visual* AddGeom(Object* obj, dGeomID geom, dReal red, dReal green, dReal blue)
{
        assert(obj && obj->numOfVisuals < kMAX_VISUALS);

        // add the geom to our geom array for the given object and
        // add it to the body

        Visual* v = &obj->visuals[obj->numOfVisuals];

        // initialize instance variables
        v->red = red;
        v->blue = blue;
        v->green = green;
        v->geom = geom;
        v->highlight = 0;
        v->drawCap = 1;
        v->alpha = 0;

        obj->numOfVisuals++;
        dGeomSetBody (geom, obj->body);
        return v;
}

//---------------------------------------------------------------------------
//  COLLISION CALLBACK FUNCTION
//
// This function is called by dSpaceCollide when two objects in space are
// potentially colliding.
static void NearCallback (void *data, dGeomID o1, dGeomID o2)
{
        int i;
        bool isTarget = false;
        bool isHand   = false;
        bool isLHand  = false;

#if kBI_MANUAL
        bool isRHand  = false;
#endif
```

```
        // Exit without doing anything if the two bodies are connected by a joint.
        // (This applies for all our tests)
        dBodyID b1 = dGeomGetBody(o1);
        dBodyID b2 = dGeomGetBody(o2);

        if (b1 && b2 && dAreConnectedExcluding(b1,b2, dJointTypeContact))
          return;

        // Ignore all contacts with body
        if ((o1 == gBody) || (o2 == gBody)) {
                return;
        }

        // Handle target as a special case
        if ((o1 == gTarget.geom) || (o2 == gTarget.geom)) {
                isTarget = true;
        }

        // Check for hand collision
        if ((o1 == lhand.g) || (o2 == lhand.g)) {
                isHand = true; isLHand = true;
        }
#if kBI_MANUAL
        else if ((o1 == rhand.g) || (o2 == rhand.g)) {
                isHand = true; isRHand = true;
        }
#endif

        // Allocate up to MAX_CONTACTS contacts per pair of objects
        dContact contact[MAX_CONTACTS];

        // Fill out the contact details, but not the geometry
        for (i=0; i < MAX_CONTACTS; i++) {

                // Contacts are bouncy and have either coulomb or infinite friction.
                // They are also a bit spongy.
                contact[i].surface.mode = dContactBounce |
                                                        (c_friction ? dContactApprox1 : 0) |
                                                        dContactSoftCFM;

                contact[i].surface.mu = (c_friction ? 0.5 : dInfinity); // friction
                contact[i].surface.mu2        = 0;       // other friction direction, not needed here
                contact[i].surface.bounce     = 0.5; // bounciness
                contact[i].surface.bounce_vel = 0.1; // minimum velocity for bounce
                contact[i].surface.soft_cfm   = 0.01; // spongyness of collision
                }

                // Fill out the contact geometry details
                int numc = dCollide(o1, o2, MAX_CONTACTS, &contact[0].geom, sizeof(dContact));

                // Have a positive collison
                if (numc > 0)
                {
                        // Ignore all collisions with target
                        if (isTarget) {
                                // If theres contact with the hand, record it and visualise it
                                if (isHand) {
#if kBI_MANUAL
                                        if (isRHand)
                                                gRec->rnear++;
                                        else
#endif
                                                gRec->lnear++;
                                        gContact = true;
                                }
                        }
                        else {
                                // For rendering, if we need to
                                dMatrix3 RI;
                                dRSetIdentity (RI);

                                // Create a contact joint for each contact...
                                for (i=0; i < numc; i++) {
                                        dJointID c = dJointCreateContact(world, contactgroup, contact+i);
                                        dJointAttach(c, b1, b2);
#if 0
                                        // ...and render it if we need to
                                        dsSetColor(0, 0, 1);
                                        if (show_contacts) {
                                                dsDrawBox(contact[i].geom.pos, RI, ss);
                                        }
#endif
                                }
                        }
                }
}



//-----------------------------------------------------------------------------
// SIM KEYPRESS CALLBACK FUNCTION
//
// Used when a key is pressed in the graphical drawstuff window.
static void Command(int cmd)
{
        // displays the view port. Replace the code in start with these lines to
        // reset the viewport
        if (cmd == 'v') {
                float xyz[3];
                float hpr[3];
                dsGetViewpoint (xyz,hpr);

                dsPrint("static float xyz[3] = {%ff, %ff, %ff};\n", xyz[0], xyz[1], xyz[2]);
                dsPrint("static float hpr[3] = {%ff, %ff, %ff};\n", hpr[0], hpr[1], hpr[2]);
        }
        else if (cmd == '2') {
                // change view to side
                float xyz[3] = {1.397406f, -2.129045f, 1.630001f};
                float hpr[3] = {111.500000f, -26.500000f, 0.000000f};
                dsSetViewpoint (xyz,hpr);
        }
        else if (cmd == '1') {
                // change view
                float xyz[3] = {2.898957f, -0.017567f, 1.630001f};
                float hpr[3] = {-177.000000f, -26.500000f, 0.000000f};
                dsSetViewpoint (xyz,hpr);
        }

        else if (cmd == '3') {
                // change view
                float xyz[3] = {2.898957f, -0.017567f, 1.630001f};
                float hpr[3] = {-177.000000f, -26.500000f, 0.000000f};
                dsSetViewpoint (xyz,hpr);
        }

        else if (cmd == 'q') {
                dsStop();
        }
}
```

```
//---------------------------------------------------------------------------
// Draws a geom object.
void DrawGeom (dGeomID g, const dReal *pos, const dReal *R, int show_aabb, int cap)
{
        if (!g)
                return;

        // Get geom information from somewhere
        if (!pos) {
                pos = dGeomGetPosition (g);
        }

        if (!R) {
                R = dGeomGetRotation (g);
        }

        // Figure out what type to render and
        // rendering information
        int type = dGeomGetClass (g);

        if (type == dBoxClass) {
                dVector3 sides;
                dGeomBoxGetLengths (g,sides);
                dsDrawBox (pos,R,sides);
        }
        else if (type == dSphereClass) {
                dsDrawSphere (pos, R, dGeomSphereGetRadius (g));
        }
        else if (type == dCCylinderClass) {
                dReal radius,length;
                dGeomCCylinderGetParams (g,&radius,&length);
                if (cap) {
                        dsDrawCappedCylinder (pos,R,length,radius);
                }
                else {
                        dsDrawCylinder (pos,R,length,radius);
                }
        }
        // Special case, Transforms
        else if (type == dGeomTransformClass) {
                // We don't want to render the transform
                // itself, so instead find its sub-geom,
                // transform its coords, and recursively
                // call this draw method.

                dGeomID g2 = dGeomTransformGetGeom (g);
                const dReal *pos2 = dGeomGetPosition (g2);
                const dReal *R2 = dGeomGetRotation (g2);
                dVector3 actual_pos;
                dMatrix3 actual_R;
                dMULTIPLY0_331 (actual_pos,R,pos2);
                actual_pos[0] += pos[0];
                actual_pos[1] += pos[1];
                actual_pos[2] += pos[2];
                dMULTIPLY0_333 (actual_R,R,R2);
                DrawGeom (g2,actual_pos,actual_R,0,cap);
        }
        else if (type == dRayClass) {
                // Draw Ray Replacement Code

                dVector3 origin, origindir, dir;

                //dGeomRayGet(g, origin, dir);
                origin[0] = pos[0]; origin[1] = pos[1]; origin[2] = pos[2];
                origindir[0] = 0; origindir[1] = 0; origindir[2] = 1;

                dMULTIPLY0_331(dir, R, origindir);

                dReal length = dGeomRayGetLength(g);
                for (int j=0; j<3; j++) dir[j] = dir[j]*length + origin[j];

                dsDrawLine (origin,dir);
                dsSetColor (0,0,1);

                dsDrawSphere (origin,dGeomGetRotation(g),0.01);
        }
}

//-----------------------------------------------------------------------------
// Totally random controls
dReal RandomControl (){
        return (dReal) rand() / (dReal) RAND_MAX;
}

//-----------------------------------------------------------------------------
// A control function for a rotational joint
void LeftShoulderController (float simTime, dJointID jid)
{
        static float torque = 0.0;
        // With some learning controllers, this speed / pos distinction
        // may not matter, b/c the gUpdateInterval is just some constant
        // to the learning algorithm.

        gRec->lstorque = abs(gRec->lstorque);
        if (gRec->lstorque > kMAX_TORQUE)
                gRec->lstorque = kMAX_TORQUE;

        gTorques.t1 = gTorques.t1 + (gRec->lstorque * kTIME_STEP);
        dJointAddHingeTorque(jid, gTorques.t1);
}

//-----------------------------------------------------------------------------
// A control function for a rotational joint
void RightShoulderController (float simTime, dJointID jid)
{

        // With some learning controllers, this speed / pos distinction
        // may not matter, b/c the gUpdateInterval is just some constant
        // to the learning algorithm.
#if kBI_MANUAL
        gRec->rstorque = abs(gRec->rstorque);
        if (gRec->rstorque > kMAX_TORQUE) {
                gRec->rstorque = kMAX_TORQUE;
        }
        dJointAddHingeTorque(jid, gRec->rstorque);
#endif
}

//-----------------------------------------------------------------------------
// A control function for a rotational joint
void LeftElbowController (float simTime, dJointID jid)
{
        // With some learning controllers, this speed / pos distinction
        // may not matter, b/c the gUpdateInterval is just some constant
        // to the learning algorithm.
        gRec->letorque = abs(gRec->letorque);
        if (gRec->letorque > kMAX_TORQUE) {
                gRec->letorque = kMAX_TORQUE;
        }
        gTorques.t2 = gTorques.t2 + (gRec->letorque * kTIME_STEP);
        dJointAddHingeTorque(jid, gTorques.t2);
}

//------------------------------------------------------------------------------
// A control function for a rotational joint
void RightElbowController (float simTime, dJointID jid)
{
        // With some learning controllers, this speed / pos distinction
        // may not matter, b/c the gUpdateInterval is just some constant
        // to the learning algorithm.
#if 0
        if (gRec->retorque > 0) {
                dJointSetHingeParam(jid, dParamVel, -1.0 / gUpdateInterval);  // velocity
        }
        else {
                dJointSetHingeParam(jid, dParamVel, 1.0 / gUpdateInterval);   // velocity
        }
        dJointSetHingeParam(jid, dParamFMax, abs(gRec->retorque));
#elif kBI_MANUAL
        gRec->retorque = abs(gRec->retorque);
        if (gRec->retorque > kMAX_TORQUE) {
                gRec->retorque = kMAX_TORQUE;
        }
        dJointAddHingeTorque(jid, gRec->retorque);
#endif
}

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

float AttachArm (dReal x, dReal y, dReal z, bool is_left)
{
        int   modifier = 1;
        float total_arm_length = 0.0;

        if (is_left == true) {
                modifier = -1;
        }

        dReal box_y_mid            = (y / 2.0);
        dReal box_z_mid        = kTARGET_HEIGHT;

        dReal arm_radius       = 0.05;          // arms are same radius
        dReal arm_mass             = 0.5;               // arms are quite light

        dReal upper_arm_length   = 0.8;
        dReal upper_arm_midpoint = (upper_arm_length / 2.0) + arm_radius;
    dReal upper_total_length = upper_arm_length  + (arm_radius * 2.0);          // add two end caps

        dReal lower_arm_length = 0.6;
        dReal lower_arm_midpoint = (lower_arm_length / 2.0) + arm_radius;
        dReal lower_total_length = lower_arm_length  + (arm_radius * 2.0);              // add two end caps

        dReal           hand_mass        = 0.2;         // light hands
    dMass               m;
        dMatrix3        matrix;
        dGeomID         g;
        Object*         obj1, *obj2, * obj3;

        //--- Create Upper Arm
        {
                //--- create upper_arm
                dMassSetCappedCylinderTotal(&m, arm_mass, 1, arm_radius, upper_arm_length);
                obj1 = NewObject();
                obj1->body = dBodyCreate(world);
                dBodySetMass (obj1->body,&m);

                dRFromAxisAndAngle (matrix, 1, 0, 0, -M_PI/2.0);                // rotate to be 90 degrees on the x axis
                dBodySetRotation(obj1->body, matrix);

                dBodySetPosition (obj1->body, 0.0, (box_y_mid + upper_arm_midpoint) * modifier, box_z_mid);

                dGeomID g = dCreateCCylinder(space, arm_radius, upper_arm_length);
                AddGeom(obj1, g, 1, 1, 0);                      // upper is violet

                //--- Attach hinge between box and arm
                dJointID hinge;

                // Attach arms to the world via hinge
                // NOTE: ODE defines the direction of the hinge via the ordering of the
                // bodies in the attach call.
                if (is_left) {
                        hinge = NewJoint (LeftShoulderController);
                        dJointAttach (hinge, obj1->body,0);                     // reverse direction
                } else {
                        hinge = NewJoint(RightShoulderController);
                        dJointAttach (hinge, 0, obj1->body);
                }

                // point in world coordinates of the joint this is where the bodies meet
                dJointSetHingeAnchor (hinge, 0.0, box_y_mid * modifier, box_z_mid);

                 // axis is along the z
                dJointSetHingeAxis (hinge, 0, 0, 1);

                // Set the strength of the hinge motor.
                // If the torque is set to 0 this essentially turns off the motor.
                dJointSetHingeParam(hinge, dParamFMax, 0.0); // torque

                // Upper arms have the same limit stops approx 220 deg of freedom
                if (is_left) {
                        dJointSetHingeParam(hinge, dParamLoStop, -0.5 * kPI);
                        dJointSetHingeParam(hinge, dParamHiStop, 0.7 * kPI);
                }
                else{
                        dJointSetHingeParam(hinge, dParamLoStop, -0.7 * kPI);
                        dJointSetHingeParam(hinge, dParamHiStop, 0.5 *kPI);
                }

                // 'power' joint to make it a little stiff
                dJointSetHingeParam(hinge, dParamFMax, 0.1);
        }


        {
                //--- Create lower arm
                obj2 = NewObject();

                obj2->body = dBodyCreate(world);
                dBodySetMass (obj2->body, &m);
                dBodySetRotation(obj2->body, matrix);
                dBodySetPosition (obj2->body, 0.0, (box_y_mid + upper_total_length + lower_arm_midpoint) * modifier, \
                                                box_z_mid);

                g = dCreateCCylinder(space, arm_radius, lower_arm_length);
                AddGeom(obj2, g, 1, 1, 0);

                //--- Attach hinge between arm and arm
                dJointID hinge2;

                if (is_left) {
                        hinge2 = NewJoint (LeftElbowController);
                        dJointAttach (hinge2, obj2->body, obj1->body);          // reverse joint
                } else {
                        hinge2 = NewJoint(RightElbowController);
                        dJointAttach (hinge2, obj1->body, obj2->body);
                }

                // point in world coordinates of the joint this is where the bodies meet
                dJointSetHingeAnchor (hinge2, 0.0, (box_y_mid + upper_total_length) * modifier, box_z_mid);

                 // axis is along the z
                dJointSetHingeAxis (hinge2, 0, 0, 1);

                // Set the strength of the hinge motor.
                // If the torque is set to 0 this essentially turns off the motor.
                dJointSetHingeParam(hinge2, dParamFMax, 0.0); // torque

                //if (is_left) {
                //      dJointSetHingeParam(hinge2, dParamLoStop, -kPI);
                //      dJointSetHingeParam(hinge2, dParamHiStop, 0.0001);                      // can't use 0 degs
                //}
                //else {
                        dJointSetHingeParam(hinge2, dParamLoStop, 0.0001);
                        dJointSetHingeParam(hinge2, dParamHiStop, kPI);
                //}
                dJointSetHingeParam(hinge2, dParamFMax, 0.1);
        }

        {
                //--- Create 'hand' -----
                obj3 = NewObject();

                // create mass
                dMassSetSphereTotal (&m, hand_mass, kHAND_SIZE);

                // create body
                obj3->body = dBodyCreate(world);
                dBodySetMass (obj3->body, &m);
                dReal yDistance = box_y_mid + upper_total_length + lower_total_length;
                dBodySetPosition (obj3->body, 0.0, (yDistance + (kHAND_SIZE/2) ) * modifier,  box_z_mid);

                // create geom
                g = dCreateBox (space, kHAND_SIZE, kHAND_SIZE, kHAND_SIZE);
                AddGeom(obj3, g, 0,1,1); // blueish

                // -- Create fixed hinge for hand
                dJointID hinge3 = dJointCreateHinge (world, 0);
                dJointAttach (hinge3, obj2->body, obj3->body);

                // point in world coordinates of the joint this is where the bodies meet
                dJointSetHingeAnchor (hinge3, 0.0, (yDistance * modifier), box_z_mid);

                 // axis is along the y
                dJointSetHingeAxis (hinge3, 0, 1, 0);

                dJointSetHingeParam(hinge3, dParamFMax, 10.0); // torque


                // record hand
                if (is_left) {
                        lhand.body = obj3->body;
                        lhand.g    = g;
```

```
                        lhand.x    =  0.0;
                        lhand.y    = (yDistance + (kHAND_SIZE/2) ) * modifier,
                        lhand.z    = box_z_mid;
                }
#if kBI_MANUAL
                else {
                        rhand.body = obj3->body;
                        rhand.g    = g;
                        rhand.x    =  0.0;
                        rhand.y    = (yDistance + (kHAND_SIZE/2) ) * modifier,
                        rhand.z    = box_z_mid;
                }
#endif

                total_arm_length = yDistance + kHAND_SIZE;
        }
        return total_arm_length;
}

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

bool BuildRobot ()
{
        gUpdateTime = 0;                                        // Reset our current update count

        // Create 'body' for agent
        gBody = dCreateBox (space, b_x, b_y, b_z);

#if kBI_MANUAL
        //--- Arm 1 (right)
        AttachArm(b_x, b_y, b_z, false);
#endif

        //--- Arm 2
        gTarget.arm_length = AttachArm(b_x, b_y, b_z, true);

        return false;
}
```

Genetics.h

```
/*
 *  Genetics.h
 *  DOD
 */

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

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

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

/**
 * Recombination probability at each locus
 * May need to be higher apporx = 0.9 which ensures virtual elitism otherwise
 * genotype gets too fragmented
 */
#define kPROB_REC 0.6

//  (2 x Input weights) + (Time & Bias for each Nodes) + Weights + (2 x Motor Gain)     + (1 synergy value)
#define kGENOTYPE_LENGTH ( (2 * kCTRNN_NUM_NODES)  + (kCTRNN_NUM_NODES + kCTRNN_NUM_NODES) + (kCTRNN_NUM_NODES * kCTRNN_NUM_NODE


// Define min/max weights of a node
#define kMAX_WEIGHT_SIZE 8.0
#define kMIN_WEIGHT_SIZE -8.0

// Define min/max time constant of a node
#define kMAX_TIME_SIZE 4.0
#define kMIN_TIME_SIZE 0.0

// Define min/max bias constant of a node
#define kMAX_BIAS_SIZE 3.0
#define kMIN_BIAS_SIZE -3.0

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

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

// Scalars on syngergy value
#define kMAX_SYNERGY_GAIN 5.0
#define kMIN_SYNERGY_GAIN -1.0

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

typedef std::vector<double>                     Genotype;                        // A single genotype
typedef std::vector<Genotype*>          Population;                     // A collection of genotypes

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

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


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

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

Genetics.cpp

```
/*
 *  Genetics.cpp
 *  DOD
 */

#include <string>
#include <vector>
#include <sstream>
#include <iostream>
```

```
#include <time.h>
#include <math.h>

#include "Globals.h"
#include "Genetics.h"
#include "Random.h"
#include "Utils.h"

static long idum = static_cast<long>(-time(0)); // seed

//-------------------------------------------------------------
// All real values ranged between 0 & 1
//-------------------------------------------------------------
#define kMAX_GENE_VALUE 1.0
#define kMIN_GENE_VALUE 0.0

void RandomiseGenotype (Genotype& gene);
void PrintGenotype     (std::ostream& out, Genotype& gene);

//-------------------------------------------------------------
// Randomises the population
void RandomisePopulation (Population& pop, int pop_size)
{
        pop.reserve(pop_size);

        for (int i = 0; i < pop_size; i++) {
                Genotype* gene = new Genotype();
                if (gene) {
                        RandomiseGenotype(*gene);
                        pop.push_back(gene);
                }
        }
}

//-------------------------------------------------------------
// Destroy population - clear out genotypes
void ClearPopulation (Population& pop)
{
        while (!pop.empty()) {
                Genotype *pGene = pop.back();
                pop.pop_back();
                delete pGene;
        }
}

//-------------------------------------------------------------
void PrintPopulation (Population& pop)
{
        Population::iterator pos;
        int counter = 0;
        for (pos = pop.begin(); pos != pop.end(); ++pos, ++counter) {
                Genotype* gene = (*pos);
                printf("[%d] ", counter);
                PrintGenotype(*gene);
                printf("\n");
        }
}

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

void RandomiseGenotype (Genotype& gene)
{
        gene.reserve(kGENOTYPE_LENGTH);
        // Randomise the sensor weights
        float inputs = 2 * kCTRNN_NUM_NODES;
        for (int i=0; i < inputs; i++) {
                gene.push_back(drand48());
        }

        // now for each gene randomise
        for (int i=0; i < kCTRNN_NUM_NODES; i++) {
                gene.push_back(drand48());              // randomised time
                gene.push_back(drand48());              // randomised bias

                for (int j=0; j < kCTRNN_NUM_NODES; j++) {
                        gene.push_back(drand48());      // randomised weights
                }
        }

        // Two motor gains
        gene.push_back(drand48());
        gene.push_back(drand48());

        // Synergy value
        gene.push_back(drand48());
}

//--------------------------------------------------------------
// Mutates a gene within a given range
// All real numbers in range [0:1]
double MutateGene (double value)
{
        Ran1(&idum);
        //mutate via guassian distribution
        double mutation = Gasdev(&idum) * GAUSVECMUT;
        value += mutation;

        // TODO: Handle saturation better?
        // Use reflection etc
        if (value < kMIN_GENE_VALUE) {
                value = kMIN_GENE_VALUE;
        }
        else if (value > kMAX_GENE_VALUE) {
                value = kMAX_GENE_VALUE;
        }
        return value;
}

//--------------------------------------------------------------
// Copy/Mutate a point in the genotype
void MutateLocus (Genotype& winner, Genotype& loser, int pos)
{
        if (drand48() < kPROB_REC) {                                            // with some prob of copying from Winner
                loser[pos] = winner[pos];
        }

        if (drand48() < kPROB_REAL_MUT) {
                loser[pos] = MutateGene(loser[pos]);
        }
}

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

void MutateGenotype (Genotype& winner, Genotype& loser)
{
        int counter = 0;

        // Do sensor weights
        float inputs = 2 * kCTRNN_NUM_NODES;
        for (int i=0; i < inputs; i++) {
                MutateLocus(winner, loser, counter++);
        }

        for (int i=0; i < kCTRNN_NUM_NODES; i++) {       // now work along genotype of Loser

                // Do time
                MutateLocus(winner, loser, counter++);

                // Do bias
                MutateLocus(winner, loser, counter++);

                for (int j = 0; j < kCTRNN_NUM_NODES; j++) {
                        // Do weights
                        MutateLocus(winner, loser, counter++);
                }
        }


        // Do motor gain
        MutateLocus(winner, loser, counter++);

        // Do motor gain
        MutateLocus(winner, loser, counter++);

        // Do synergy
        MutateLocus(winner, loser, counter++);
}

//--------------------------------------------------------------
void PrintGenotype (Genotype& gene)
{
        PrintGenotype(std::cout, gene);
}

//--------------------------------------------------------------
std::string     GenotypeToString        (Genotype& gene)
{
        std::stringstream str;
        PrintGenotype(str, gene);
        return str.str();
}

//--------------------------------------------------------------
std::string GenotypeToDot               (Genotype& gene)
{
        std::stringstream str;

        // print to 2 decimal places
        str.precision(2);
        str << "digraph Genotype {" << std::endl;

        // export input's
        str << "\tsubgraph cluster1 {" << std::endl;
        str << "\t\tsensor1 [shape = point, style = filled];" << std::endl;
        str << "\t\tsensor2 [shape = point, style = filled];" << std::endl;
        str << "\t\tlabel=\"inputs\"" << std::endl << "}" << std::endl;

        str << "\tsubgraph cluster2 {" << std::endl;
        str << "\t\tmotor1  [shape = point, style = filled];" << std::endl;
        str << "\t\tmotor2  [shape = point, style = filled];" << std::endl;
        str << "\t\tlabel=\"outputs\"" << std::endl << "}" << std::endl;

        int counter = 0;
        int smotor1 = kCTRNN_NUM_NODES - 1;
        int smotor2 = kCTRNN_NUM_NODES - 2;
        int emotor1 = kCTRNN_NUM_NODES - 3;
        int emotor2 = kCTRNN_NUM_NODES - 4;

        float inputgain1 = RescaleUnitValue(kMIN_INPUT_GAIN, kMAX_INPUT_GAIN, gene[0]);
        float inputgain2 = RescaleUnitValue(kMIN_INPUT_GAIN, kMIN_INPUT_GAIN, gene[1]);

        counter = (2 * kCTRNN_NUM_NODES);

        str << "\tsensor1 -> N0 [label=\"" << inputgain1 << "\"]" << std::endl;
        str << "\tsensor2 -> N1 [label=\"" << inputgain2 << "\"]" << std::endl;

        str << "\tnode [fontsize=8]" << std::endl;
        str << "\tedge [fontsize=8]" << std::endl;

        // now for each gene randomise
        for (int i=0; i < kCTRNN_NUM_NODES; i++) {
                float time = pow(10, RescaleUnitValue(kMIN_TIME_SIZE, kMAX_TIME_SIZE, gene[counter++]));
                float bias = RescaleUnitValue(kMIN_BIAS_SIZE, kMAX_BIAS_SIZE, gene[counter++]);

                str << "\tN" << i << " [shape=record, style=rounded, label=\"{T=" << time << " | N" << i << " | B=" << bias << "
                for (int j=0; j < kCTRNN_NUM_NODES; j++) {
                        double weight = RescaleUnitValue(kMIN_WEIGHT_SIZE, kMAX_WEIGHT_SIZE, gene[counter++]);
                        double aweight = fabs(weight);
                        str << "\tN" << i << "-> N" << j << " [weight=" << aweight << " style=\"setlinewidth(" << aweight;
                        str << ")\", label=\"" << weight << "\"";

                        // mark large weights with larger arrows
                        if (aweight > 4.0) {
                                str << ", arrowsize=2";
                        }

                        str << "]" << std::endl;
                }
        }

#if kENFORCE_SHOULDER_ELBOW_SYM
        float weight_gain = RescaleUnitValue(kMIN_MOTOR_GAIN, kMAX_MOTOR_GAIN, gene[counter++]);
        str << "\tN" << emotor1 << " -> motor1 [label=\""  << weight_gain << "\"]" << std::endl;
        weight_gain = RescaleUnitValue(kMIN_MOTOR_GAIN, kMAX_MOTOR_GAIN, gene[counter++]);
        str << "\tN" << emotor2 << " -> motor2 [label=\""  << weight_gain << "\"]" << std::endl;
#endif
        str << "}" << std::endl;
        return str.str();
}

//-------------------------------------------------------------
void PrintGenotype (std::ostream& out, Genotype& gene)
{
        int counter = 0;

        // Print out sensor weights
        out.precision(3);

        float two_out = 2 * kCTRNN_NUM_NODES;
        out << "<";
        for (int i=0; i < two_out; i++) {
                out << " " << gene[counter++];
        }
        out << ">" << std::endl;

        // Print time and bias
        for (int i=0; i < kCTRNN_NUM_NODES; i++) {
                out << std::endl;
                out << "(t=" << pow(10, RescaleUnitValue(kMIN_TIME_SIZE, kMAX_TIME_SIZE, gene[counter++]));
                out << "\tb=" <<  RescaleUnitValue(kMIN_BIAS_SIZE, kMAX_BIAS_SIZE, gene[counter++]) << "\t";

                // Print connection weights
                out << "[";
                for (int j=0; j < kCTRNN_NUM_NODES; j++) {
                        out << RescaleUnitValue(kMIN_WEIGHT_SIZE, kMAX_WEIGHT_SIZE, gene[counter++]) << "\t";
                }
                out << "]";
        }

        out << " m1=" << RescaleUnitValue(kMIN_MOTOR_GAIN, kMAX_MOTOR_GAIN, gene[counter++]);
        out << " m2=" << RescaleUnitValue(kMIN_MOTOR_GAIN, kMAX_MOTOR_GAIN, gene[counter++]);

        out << " S=" << gene[counter++];
}
```

Globals.h

```
/*
 *  Globals.h
 *  DOD
 */

#pragma once

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


/**
 * If true, build a bimanual robot
 */
#define kBI_MANUAL 0

/**
 * Size of the population to evaluate
 */
#define kNUM_POPULATION 40


/**
 * How many outer loops of the Microbial GA to perform
 */
#define kNUM_GENERATIONS 10

/**
 * How many targets to evaluate
 */
#define kNUMBER_OF_TARGETS_PER_TRIAL 5

/**
 * How long each trial is in time steps
 */
#define kMIN_LENGTH_OF_TRIAL 800
#define kMAX_LENGTH_OF_TRIAL 1400

/**
 * Number of CTRNN nodes
 */
#define kCTRNN_NUM_NODES 8
```

```
/**
 * Run CTRNN connected to blank world
 */
#define kNUMBER_OF_PREEVAL_STEPS 50

/**
 *
 */
#define kTIME_STEP 0.01

/**
 * Define own version of PI
 */
#define kPI 3.14159265358979323846264338327950

/**
 *      Size of 'hands'
 */
#define kHAND_SIZE 0.1

/**
 * How near hand has to be score fitness
 * (if used)
 */
#define kHAND_NEAR_THRESHOLD (2 * kHAND_SIZE)

/**
 * Appies symmetry between shoulder and elbow
 */
#define kENFORCE_SHOULDER_ELBOW_SYM 1

/**
 * Should we try and replace hands at the end of every target
 */
#define kRESET_HANDS 0

/**
 * Set alpha and beta modifiers in fitness function
 */
#define kALPHA 1.0                      //!< Distance to target
#define kBETA  0.0                      //!< Contact time with target
```

Logs.h

```
/*
 *  Logs.h
 *  DOD
 */

#pragma once

void SaveFitness (std::string fileName, float* vect1, float* vect2, float* vect3, int length);

void InputOutputLog     (SimulationRecord& rec);
void DistanceLog        (SimulationRecord& rec, float ldist, float rdist, float lfit, float rfit);
void PotentialLog       (SimulationRecord& rec);
void FitnessLog         (SimulationRecord& rec, float near, float contact);

void StartLogs          ();
void EndLogs            ();
```

Logs.cpp

```
/*
```

```
 *  Logs.cpp
 *  DOD
 */

#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
using namespace std;

#include "Globals.h"
#include "Genetics.h"
#include "Random.h"
#include "SimulationRecord.h"
#include "Simulation.h"

#include "Utils.h"
#include "Logs.h"

#define kIO_LOG                                         "IOLog.txt"
#define kDEFAULT_RUN_FITNESS_FILE       "RunFitness.txt"
#define kDISTANCE_LOG                           "DistanceLog.txt"
#define kPOTENTIAL_LOG                          "PotentialLog.txt"
#define kFITNESS_LOG                            "FitnessLog.txt"

std::ofstream* gFitnessLog  = 0;
std::ofstream* gIOLog       = 0;                                // log input and torque values for arms
std::ofstream* gDistanceLog = 0;                                // log distance values for robot
std::ofstream* gPotLog      = 0;                                // log cell potentials

//----------------------------------------------------------------------
// Logging Code
//----------------------------------------------------------------------

void StartLogs()
{
        gIOLog       = new ofstream();
        gDistanceLog = new ofstream();
        gPotLog      = new ofstream();
        gFitnessLog  = new ofstream();

        try {
                std::string fileName = kDEFAULT_FILE_PATH;
                fileName += kIO_LOG;

                (*gIOLog).open(fileName.c_str(), ios::out);
                if ((*gIOLog).is_open()) {
                        (*gIOLog) << "#plot \"" << fileName.c_str() << "\"";
                        (*gIOLog) << " using 1:2 title \'shoulder\' with lines, \"";
                        (*gIOLog) << fileName.c_str() << "\" using 1:3 title \'elbow\' with lines, \"";
                        (*gIOLog) << fileName.c_str() << "\" using 1:4 title \'input-x\' with lines, \"";
                        (*gIOLog) << fileName.c_str() << "\" using 1:5 title \'input-y\' with lines";
                        (*gIOLog) << std::endl;
                        (*gIOLog) << "#plot \"" << fileName.c_str() << "\"";
                        (*gIOLog) << " using 2:3 title \'phase\' with lines" << std::endl;
                        (*gIOLog) << "#plot \"" << fileName.c_str() << "\"";
                        (*gIOLog) << " using 1:4 title \'input\' with lines" << std::endl;
                }

                fileName = kDEFAULT_FILE_PATH;
                fileName += kDISTANCE_LOG;

                (*gDistanceLog).open(fileName.c_str(), ios::out);
                if ((*gDistanceLog).is_open()) {
                        (*gDistanceLog) << "#plot \"" << fileName.c_str() << "\"";
                        (*gDistanceLog) << " using 1:2 title \'Left\' with lines, \"";
                        (*gDistanceLog) << fileName.c_str() << "\" using 1:3 title \'Right\' with lines" << endl;
                        (*gDistanceLog) << "#plot \"" << fileName.c_str() << "\"";
                        (*gDistanceLog) << " using 1:4 title \'lfit\' with lines, \"";
                        (*gDistanceLog) << fileName.c_str() << "\" using 1:5 title \'rfit\' with lines";
                        (*gDistanceLog) << endl;
                }

                fileName = kDEFAULT_FILE_PATH;
                fileName += kPOTENTIAL_LOG;

                (*gPotLog).open(fileName.c_str(), ios::out);
                if ((*gPotLog).is_open()) {
                        (*gPotLog) << "#plot \"" << fileName.c_str() << "\"";
                        (*gPotLog) << " using 1:2 title \'0\' with lines, \"";
                        (*gPotLog) << fileName.c_str() << "\" using 1:3 title \'1\' with lines, \"";
                        (*gPotLog) << fileName.c_str() << "\" using 1:3 title \'2\' with lines, \"";
                        (*gPotLog) << fileName.c_str() << "\" using 1:4 title \'3\' with lines, \"";
                        (*gPotLog) << fileName.c_str() << "\" using 1:5 title \'4\' with lines, \"";
                        (*gPotLog) << fileName.c_str() << "\" using 1:6 title \'5\' with lines, \"";
                        (*gPotLog) << fileName.c_str() << "\" using 1:7 title \'6\' with lines";
                        (*gPotLog) << endl;
                }

                fileName = kDEFAULT_FILE_PATH;
                fileName += kFITNESS_LOG;

                (*gFitnessLog).open(fileName.c_str(), ios::out);
                if ((*gFitnessLog).is_open()) {
                        (*gFitnessLog) << "#plot \"" << fileName.c_str() << "\"";
                        (*gFitnessLog) << " using 1:2 title \'0\' with lines";
                        (*gFitnessLog) << endl;
                }
        }
        catch(...) {
        }
}

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

void EndLogs()
{
        (*gIOLog).close();
        (*gDistanceLog).close();
        (*gPotLog).close();
        (*gFitnessLog).close();

        delete gDistanceLog;
        gDistanceLog = 0;

        delete gIOLog;
        gIOLog = 0;

        delete gPotLog;
        gPotLog = 0;

        delete gFitnessLog;
        gFitnessLog = 0;
}

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

void InputOutputLog (SimulationRecord& rec)
{
        if (gIOLog && gIOLog->is_open()) {
                (*gIOLog) << rec.total_steps << "\t";
                (*gIOLog) << rec.lstorque << "\t";
                (*gIOLog) << rec.letorque << "\t";
                (*gIOLog) << rec.inputs[0] << "\t" << rec.inputs[1] << "\t" << rec.inputs[2];

                (*gIOLog) << endl;
        }
}

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

void FitnessLog (SimulationRecord& rec, float near, float contact)
{
        if (gFitnessLog && gFitnessLog->is_open()) {
                (*gFitnessLog) << (rec.targets - 1) << "\t";
                (*gFitnessLog) << near << "\t";
                (*gFitnessLog) << contact << "\t";

                (*gFitnessLog) << endl;
        }
}

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

void PotentialLog (SimulationRecord& rec)
{
        if (gPotLog && gPotLog->is_open()) {
                (*gPotLog) << rec.total_steps << "\t";

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

                (*gPotLog) << "\t" << rec.inputs[0];
                (*gPotLog) << "\t" << rec.inputs[1];

                (*gPotLog) << endl;
        }
}

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

void DistanceLog (SimulationRecord& rec, float ldist, float rdist, float lfit, float rfit)
{
        if (gDistanceLog && gDistanceLog->is_open()) {
                (*gDistanceLog) << rec.total_steps << "\t";
                (*gDistanceLog) << ldist << "\t";
                (*gDistanceLog) << rdist << "\t";
                (*gDistanceLog) << lfit << "\t" << rfit;

                (*gDistanceLog) << endl;
        }
}

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

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

        fit_file << "#plot \"" << fileName << "\"";
        fit_file << " using 1:2 title 'worst' with lines, \"";
        fit_file << fileName << "\" using 1:3 title 'avg' with lines, \"";
        fit_file << fileName << "\" using 1:4 title 'best' with lines" << std::endl;
        fit_file << "#Worst\tAvg\tBest" << endl;

        FloatVector::iterator pos;
        for (int i = 0; i < length; i++) {
                fit_file << i << "\t" << vect1[i];

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

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

Random.h

```
/*
 *  Random.h
 *  DOD
 */

#pragma once

/************************************************************************
* name:         Ran1 & Gasdev
* description:  Random & Gaussian random generators from
*               Random Number Generators from Numerical Recipes in C
*               v2.0 (C) Copr. 1986-92 Numerical Recipes Software
************************************************************************/
float Ran1      (long *idum);
float Gasdev(long *idum);
```

Random.cpp

```
/*
 *  Random.cpp
 *  DOD
 */

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

//---------------- random generator parameters
#define IA       16807
#define IM       2147483647
#define AM       (1. /IM)
#define IQ       127773
#define IR       2836
#define NTAB    32
#define NDIV    (1+(IM-1)/NTAB)
#define EPS      1.2e-7
#define RNMX    (1.0-EPS)

/************************************************************************
* name:         Ran1 & Gasdev
* description:  Random & Gaussian random generators from
*               Random Number Generators from Numerical Recipes in C
*               v2.0 (C) Copr. 1986-92 Numerical Recipes Software
************************************************************************/
```

```
float Ran1 (long *idum){
        int j;
        long k;
        static long iy=0;
        static long iv[NTAB];
        float temp;

        if (*idum <= 0 || !iy) {
                  if (-(*idum) < 1)
                        *idum = 1;
                  else *idum = -(*idum);

                  for (j = NTAB + 7; j >= 0; j--) {
                          k = (*idum)/IQ;
                          *idum = IA*(*idum-k*IQ)-IR*k;
                          if (*idum <0)
                                        *idum += IM;
                          if (j < NTAB)
                                        iv[j] = *idum;
                  }
                  iy = iv[0];
        }
        k = (*idum)/IQ;
        *idum=IA*(*idum-k*IQ)-IR*k;
        if (*idum < 0)
                *idum += IM;
        j = iy/NDIV;
        iy=iv[j];
        iv[j] = *idum;
        if ((temp=AM*iy) > RNMX)
                return RNMX;
        else
                return temp;
}

//--------------------------------------------------------------------
float Gasdev(long *idum)
{
        static int iset=0;
        static float gset;
        float fac,rsq,v1,v2;

        if (iset == 0) {
                do {
                        v1 = 2.0 * Ran1(idum)-1.0;
                        v2 = 2.0 * Ran1(idum)-1.0;
                        rsq= (v1*v1) + (v2*v2);
                }
                while (rsq >= 1.0 || rsq==0.0);

                fac = sqrt(-2.0*log(rsq)/rsq);
                gset= v1 * fac;
                iset= 1;
                return v2*fac;
        }
        iset = 0;
        return gset;
}
```

Simulation.h

```
/*
 *  Simultion.h
 *  DOD
 */
```

```
class SimulationRecord;

/**
 * Start evolutionary process
 */
void Evol                                       (void);

/**
 * Evaluate a previous saved genotyp
 */
void Eval (std::string fileName);

/**
 * Loads the default fitness case
 */
void EvalFitest();

/**
 * Integrate controller a single step
 * @param gRec  Current simulaton record
 * @param x1    Position of the left hand (x)
 * @param y1    Position of the left hand (y)
 * @param x2    Position of the right hand (x)
 * @param y2    Position of the right hand (y)

 * @param tx    Position of the target (x)
 * @param ty    Position of the target (y)
 */
bool IntegrateSingleStep (SimulationRecord* gRec, float x1, float y1,
                                                                        float x2, float y2, float tx, float ty);


/**
 * Callback to set fitness
 */
void CalcFitness (SimulationRecord* rec);
```

Simulation.cpp

```
/*
 *  Simultion.cpp
 *  DOD
 */

#include <algorithm>
#include <vector>
#include <math.h>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <time.h>

#if WIN_ENV
#include <float.h>                     // required for _isnan
#endif
using namespace std;

#include "Globals.h"

#include "AgentModel.h"
#include "Genetics.h"
#include "Random.h"
#include "Simulation.h"
```

```
#include "Utils.h"
#include "Logs.h"

#define kDEFAULT_END_AGENT_FILE         "Agent.txt"
#define kDEFAULT_FITNESS_FILE           "EvoFitness.txt"

static void      SavePopulationMember (std::string fileName, Genotype& gene);
static Genotype* LoadPopulationMember (std::string fileName);
static void      ExportPopulationMemberAsDot (std::string fileName, Genotype& gene);

static float     EvaluateGenotype (Genotype& gene, bool visualise = false);

//----------------------------------------------------------------------
// look-up table for sigmoid function
double SIGMOID[10000];


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

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

void CalcFitness (SimulationRecord* rec)
{
        if (rec == NULL)
                return;

        float near    = (rec->laccumulator/rec->max_steps);
        float contact = (rec->lnear/rec->max_steps);

        FitnessLog(*rec, near, contact);
        rec->fitness[rec->targets - 1] = (kALPHA * near) + (kBETA * contact);
}

//----------------------------------------------------------------------
// Run the CTRNN for a few steps
bool PreIntegrate (SimulationRecord& rec)
{
        bool ok = true;
        vector<double> sig (kCTRNN_NUM_NODES);

        // No input
        for (int i = 0; i < kCTRNN_NUM_NODES; i++) {
                rec.inputs[i] = 0.0;
        }

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

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

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

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

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

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

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

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

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

                        // Finally if values have gone out of bound agent is failed
                        if ( isnan(rec.values[i]) ) {
                                // values are out of range
                                ok = false;
                                break;
                        }
                }
        }
        return ok;
}

//-----------------------------------------------------------------------
//
bool IntegrateSingleStep(SimulationRecord* rec, float x1, float y1, float x2, float y2, float tx, float ty)
{
        if (rec == NULL)
                return false;

        // distance between left hand and target
        float ldistance = DistanceBetweenPoints(x1, y1, tx, ty);

        // distance between right hand and target
        float rdistance = DistanceBetweenPoints(x2, y2, tx, ty);

        // Single input neuron model
#if 1
                rec->inputs[0] = (tx - x1) * (rec->input_weights[0]);
                rec->inputs[1] = (ty - y1) * (rec->input_weights[1]);
                rec->inputs[2] = (rec->input_weights[2]) * rec->tangle;
#else
        for (int i = 0; i < kCTRNN_NUM_NODES; i++) {
                rec->inputs[i] = (tx - x1) * (rec->input_weights[i]);
        }

        for (int i = kCTRNN_NUM_NODES; i < ( 2* kCTRNN_NUM_NODES); i++) {
                rec->inputs[i-kCTRNN_NUM_NODES] += rec->tangle * (rec->input_weights[i]);
        }
#endif

        vector<double> sig (kCTRNN_NUM_NODES);
        {
                // integrate network in this time step
                for (int i= 0; i < kCTRNN_NUM_NODES; i++) {
                        // -yi
                        double change = -1.0 * rec->values[i];

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

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

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

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

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

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

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

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

                // generate motor values
                // NOTE: Last two nodes in genotype are motor effectors
                if (!rec->failed)
                {
                        assert(kCTRNN_NUM_NODES >= 6);

                        // Get firing rate for neurons
                        int smotor1 = kCTRNN_NUM_NODES - 1;
                        int smotor2 = kCTRNN_NUM_NODES - 2;
                        int emotor1 = kCTRNN_NUM_NODES - 3;
                        int emotor2 = kCTRNN_NUM_NODES - 4;

                        // Scale by evolvable motor values
                        // Have antagnonistic neurons

#if kENFORCE_SHOULDER_ELBOW_SYM
                        rec->letorque = (rec->motor_gain2) * (rec->values[emotor1] - rec->values[emotor2]);
                        rec->lstorque = rec->syn * rec->letorque;
#else
                        rec->lstorque = (rec->motor_gain1) * (rec->values[smotor1] - rec->values[smotor2]);
                        rec->letorque = (rec->motor_gain2) * (rec->values[emotor1] - rec->values[emotor2]);
#endif
                        // Log info
                        InputOutputLog(*rec);
                        PotentialLog(*rec);
                }
                rec->steps++;
                rec->total_steps++;
        }

        //--- record distance to target and fitness
        float ldist = fabs(ldistance);
        float rdist = fabs(rdistance);
        float rfit  = 0.0;
        float lfit  = 1.0 - (ldist / rec->start_ldistance);
        //if (ldist <= rec->start_ldistance) {
                rec->laccumulator += lfit;
        //}
#if 0
        if (ldist < kHAND_NEAR_THRESHOLD) {
                rec->lnear++;
        }
#endif


#if kBI_MANUAL
        rfit  = 1.0 - (rdist / rec->start_rdistance);

        if (rdist <= rec->start_rdistance) {
                rec->raccumulator += rfit;
        }
#if 0
        if (rdist < kHAND_NEAR_THRESHOLD) {
                rec->rnear++;
        }
#endif
#endif
        DistanceLog(*rec, ldist, rdist, lfit, rfit);

        return (!rec->failed);
}

//----------------------------------------------------------------------
// Map genotype to phenotype
// Returns fitness score for an evaluation
float EvaluateGenotype (Genotype& gene, bool visualise)
{
        float overall = 0.0;
        vector<float> fitness (kNUMBER_OF_TARGETS_PER_TRIAL);

        SimulationRecord rec(gene);

        // Clear old run info
        rec.ClearNonGenetic();

        // Set the length of each trial
        rec.max_steps   = RangedDouble(kMIN_LENGTH_OF_TRIAL, kMAX_LENGTH_OF_TRIAL);
        rec.max_targets = kNUMBER_OF_TARGETS_PER_TRIAL;

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

        // Run model/Integrate - will run over kNUMBER_OF_TARGETS_PER_TRIAL trials
        RunModel(rec, visualise);

        // ---- Calc fitnes
        // return average fitness
        overall = FloatVectorMean(rec.fitness, kNUMBER_OF_TARGETS_PER_TRIAL);
        // Minus 20% of the standard deviation
        overall = overall - (float) (FloatVectorStdDev(rec.fitness, overall, kNUMBER_OF_TARGETS_PER_TRIAL) * 0.2);

        return overall;
}

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

void EvolInit ()
{
        // Seed random number generator
        srand(static_cast<int>(time(NULL)));

        // Eduardo's suggested method for pre-generating sigmoid
        // values
        for (int i = 0; i < 10000; i++) {
                SIGMOID[i]=1/(1+exp(-((double)(i-5000)/500)));
        }
}

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

void Evol (void)
{
        // Generate Population
        Population pop;
        RandomisePopulation(pop, kNUM_POPULATION);

        PrintPopulation(pop);

        // Run microbial Ga

        // Fitness value of each population member
        std::vector<float> fitness              (kNUM_POPULATION, 0.0);

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

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

        Genotype* aGene, *bGene, *pWinner, *pLoser;

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

                        // pick 2 at random
                        int a = static_cast<int>(kNUM_POPULATION * drand48());
                        int b = static_cast<int>(kNUM_POPULATION * drand48());

                        // Try again - but doesn't matter if they're the same
                        if ( a == b)
                                b = static_cast<int>(kNUM_POPULATION * drand48());

                        // Look for fitess genotype
                        aGene = pop[a];
                        bGene = pop[b];
                        fitness[a] = (float) EvaluateGenotype(*aGene);
                        fitness[b] = (float) EvaluateGenotype(*bGene);

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

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

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

                // ---- Display generation information
#if WIN_ENV
                _snprintf_s(buffer, 512, "Fitest = (%.2f), Avg. = %.2f - step (%d of %d)", best_fitness[gen_count], \
                                                                                avg_fitness[gen_count], count, kNUM_GENERATIONS
#else
                snprintf(buffer, 512, "\nFitest = (%.2f), Avg. = %.2f - step (%d of %d)", best_fitness[gen_count], \
                                                                                avg_fitness[gen_count], count, kNUM_GENERATIONS
#endif
                FloatVectorPrint (fitness, buffer);
        }

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

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

                // Chose the fitness to pick approx 5 fitest
                int threshold = rank[kNUM_POPULATION - 5];

                // Now loop and save fitest agents
                stringstream fileName;

                int index = 0;
                int fitest = 0;
                Genotype* gene =0;

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

                                gene = pop[n];
                                SavePopulationMember(fileName.str(), *gene);
                        }

                        if (fitness[n] > fitest) {
                                index = n;
                                fitest = fitness[n];
                        }
                }
                //---- Save Fitest ------
                // Save population member in default file
                fileName.str("");               // clear
                fileName << kDEFAULT_FILE_PATH;
                fileName << kDEFAULT_END_AGENT_FILE;

                gene = pop[index];
                SavePopulationMember(fileName.str(), *gene);

                fileName.str("");
                fileName << kDEFAULT_FILE_PATH;
                fileName << kDEFAULT_FITNESS_FILE;
                SaveFitness(fileName.str(), worst_fitness, avg_fitness, best_fitness, kNUM_GENERATIONS);
        }
        // End of GA
}

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

void EvalFitest()
{
        std::string fileName = kDEFAULT_FILE_PATH;
        fileName += kDEFAULT_END_AGENT_FILE;
        Eval(fileName);
}

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

void Eval (std::string fileName)
{
        Genotype* gene = LoadPopulationMember(fileName);
        if (gene == NULL)
                return;

        EvolInit();
        StartLogs();

        // Use visualisation for this
        EvaluateGenotype(*gene, true);
        EndLogs();

        fileName = kDEFAULT_FILE_PATH;
        fileName += "Agent.dot";

        ExportPopulationMemberAsDot(fileName, *gene);

        delete gene;
}



//---------------------------------------------------------------
// Input/Export of genotypes
//---------------------------------------------------------------

void SavePopulationMember (std::string fileName, Genotype& gene)
{
        ofstream out;
        out.open(fileName.c_str(), ios::out);

        Genotype::iterator pos = gene.begin();
        for (; pos != gene.end(); ++pos) {
                out << (*pos) << " ";
        }
        out.close();
}

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

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

        assert(in.is_open());

        Genotype* gene = new Genotype();

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

//--------------------------------------------------------------------
// Write out a genotype as a .dot file
void ExportPopulationMemberAsDot (std::string fileName, Genotype& gene)
{
        ofstream out;
        out.open(fileName.c_str(), ios::out);

        std::string outStr = GenotypeToDot(gene);
        out << outStr;
        out.close();
}
```

SimulationRecord.h

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

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

typedef std::vector<float> FloatVector;

class  SimulationRecord
{
public:
        SimulationRecord();
        SimulationRecord(Genotype& gene);

        void ClearNonGenetic();
        void ClearTemporary();

        FloatVector             ntimes;                                         //!< time values of node
        FloatVector             biases;                                         //!< bias values for nodes
        FloatVector             values;                                         //!< current value for nodes
        FloatVector             inputs;                                         //!< current input for nodes
        float           weights[kCTRNN_NUM_NODES][kCTRNN_NUM_NODES];    // weights

        // Sensory weights
        float                   input_weights[kCTRNN_NUM_NODES + kCTRNN_NUM_NODES];

        // Motor weight
        float                   motor_gain1;
        float                   motor_gain2;

        // float synergy value
        float                   syn;

        // ---- Preserved across runs
        bool                    failed;
        float                   step_size;                                  //!< integration step size
        float                   max_steps;
        int                             max_targets;                        //!< Number of targets to sim
        int                             targets;                                    //!< Target counter
        float                   fitness[kNUMBER_OF_TARGETS_PER_TRIAL];
        float                   total_steps;

        // --- Not preserved information
        float                   steps;                                      //!< counter of integration steps (this trial)

#if kBI_MANUAL
        float                   start_rdistance;                    //!< Start distance right hand is from target
        float                   raccumulator;                       //!< Distance scores for right hand
        float                   rnear;                                      //!< Steps right hand was near target

        float                   rstorque;                                   //!< Torque on right shoulder joint
        float                   retorque;                                   //!< Torque on right elbow joint
#endif

        float                   start_ldistance;                    //!< Start distance left hand is from target

        float                   laccumulator;                       //!< Distance scores for left hand
        float                   lnear;                                      //!< Steps left hand was near target

        float                   lstorque;                                   //!< Torque on left shoulder joint
        float                   letorque;                                   //!< Torque on left elbow joint

        float                   tangle;                                     //!< Angle of target from CTRNN
};
```

SimulationRecord.cpp

```
/*
 *  SimulationRecord.cpp
 *  DOD
 */

#include <math.h>
#include <vector>

#include "Globals.h"
#include "Genetics.h"
#include "SimulationRecord.h"
#include "Utils.h"

//-------------------------------------------------------------------------------------
// Load genotype in a more convenient record
SimulationRecord::SimulationRecord(Genotype& gene)
{
        ntimes.reserve(kCTRNN_NUM_NODES);
        biases.reserve(kCTRNN_NUM_NODES);
        values.reserve(kCTRNN_NUM_NODES);
        inputs.reserve(kCTRNN_NUM_NODES);
        // Read in genotype into more convenient data structure
        Genotype::iterator pos = gene.begin();

        float two_out = 2 * kCTRNN_NUM_NODES;
        float iGain, fGain, sGain;

        // Read in input weights
        for (int i = 0; i < two_out; i++) {
                iGain = (float) (*pos);                                 // record sensor gain
                input_weights[i] = RescaleUnitValue(kMIN_INPUT_GAIN, kMAX_INPUT_GAIN, iGain);   // record motor gain
                pos++;
        }

        float bias, time, weight;

        for (int i = 0; i < kCTRNN_NUM_NODES; i++) {
                // load node time value
                time = (float) (*pos);

                // linearly scale
                time = RescaleUnitValue(kMIN_TIME_SIZE, kMAX_TIME_SIZE, time);

                // now scale exponentially
#if 0
                ntimes.push_back(pow(10, time));        // record time for the node
#else
                ntimes.push_back(time); // record time for the node
#endif
                ++pos;

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

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

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

        fGain = (float)(*pos);
        motor_gain1 = RescaleUnitValue(kMIN_MOTOR_GAIN, kMAX_MOTOR_GAIN, fGain);        // record motor gain
        pos++;

        fGain = (float)(*pos);
        motor_gain2 = RescaleUnitValue(kMIN_MOTOR_GAIN, kMAX_MOTOR_GAIN, fGain);        // record motor gain

        sGain = (float)(*pos);
        syn = RescaleUnitValue(kMIN_SYNERGY_GAIN, kMAX_SYNERGY_GAIN, sGain);    // record synergy gain


        failed          = false;
        step_size       = (float) kTIME_STEP;                                   //!< integration step size
        max_steps   = 0.0;
        max_targets = targets = 0;
        total_steps = 0;
        ClearTemporary();
}

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

void SimulationRecord::ClearTemporary()
{
        steps = 0;

#if kBI_MANUAL
        start_rdistance = 0.0;
        rnear = 0.0;
        raccumulator = 0.0;
        rstorque = retorque = 0.0;
#endif

        start_ldistance = 0.0;
        laccumulator    = 0.0;
        lnear           = 0.0;

        lstorque = letorque = 0.0;
        tangle = 0.0;
}

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

void SimulationRecord::ClearNonGenetic ()
{
    ClearTemporary();
        for (int i = 0; i < kCTRNN_NUM_NODES; i++) {
                float bias = biases[i];
                values[i] = ((float) RangedDouble(-bias - 1.0, -bias + 1.0));
                inputs[i] = 0.0;
        }

        failed          = false;
        step_size       = (float) kTIME_STEP;                                   //!< integration step size
        max_steps = 0.0;
        max_targets = targets = 0;
        total_steps = 0;
        memset(&fitness, 0, kNUMBER_OF_TARGETS_PER_TRIAL * sizeof(float));
}
```

Utils.h

```
/*
 *  Utils.h
 *  DOD
 */

#pragma once

typedef std::vector<float> FloatVector;

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

float  RescaleUnitValue (float min, float max, float value);
double RangedDouble             (double min_num, double max_num);

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

```
float DistanceBetweenPoints      (float x1, float y1, float x2, float y2);

//---------------------------------------------------------------------
// Float Vector utils
void FloatVectorPrint   (float* fit, int n, const char* title);
void FloatVectorPrint   (FloatVector& fit, const char* title);

float FloatVectorMean   (float* input, int n);
float FloatVectorMean   (FloatVector& input, int n);

float FloatVectorStdDev (float* input, float mean, int n);
float FloatVectorStdDev (FloatVector& input, float mean, int n);

#if WIN_ENV
// Not provided on Windows
double drand48 ();
#endif
```

Utils.cpp

```
/*
 *  Utils.cpp
 *  DOD
 */

#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <vector>
using namespace std;

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

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

float RescaleUnitValue (float min, float max, float value)
{
        // Linear scale value between [0,1] to new range [min, max]
        return (min + (max - min) * value);
}


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

void  RotatePointAroundPoint (float& x1, float& y1, float x2, float y2, float angle)
{
        float x = x1 - y1;
        float y = y1 - y2;

        float rad =  angle * (float) (kPI/180.0);

        float x_new = (cos(rad) * x) - (sin(rad) * y);
        float y_new = (cos(rad) * y) + (sin(rad) * x);

        x1 = x_new + x2;
        y1 = y_new + y2;
}

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

float DistanceBetweenPoints(float x1, float y1, float x2, float y2)
{
        float xd = x2 - x1;
        float yd = y2 - y1;
        float distance =  sqrt( (xd * xd) + (yd * yd));

        return distance;
}

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

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

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

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


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

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

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

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

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

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

```
#endif

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

double RangedDouble (double min_num, double max_num)
{
        double val = drand48() * (max_num-min_num) + min_num;

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

        return val;
}

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

void FloatVectorPrint   (float* fit, int n, const char* title)
{
        printf("%s\n", title);
        for (int i = 0; i < n; i++) {
                printf("%.2f\t", fit[i]);
        }
        printf("\n");

}

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

}

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

float FloatVectorMean   (float* input, int n)
{
        float mean = 0.0;

        vector<float>::iterator pos;
        for (int i = 0; i < n; i++) {
                mean += input[i];
        }
        mean /= (float) n;
        return mean;
}

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

float FloatVectorMean (FloatVector& input, int n)
{
        float mean = 0.0;

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

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

float FloatVectorStdDev (float* input, float mean, int n)
{
        float sum = 0.0;

        for (int i = 0; i < n; i++) {
                sum += pow((input[i] - mean), 2);
        }
        sum /= (float) n;
        return sum;
}

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

float FloatVectorStdDev (FloatVector& input, float mean, int n)
{
        float sum = 0.0;

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