

To the Graduate Council:

I am submitting herewith a thesis written by Scott Frederick Hansen entitled “Exploration of Neural Structures for Dynamic System Control.” I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

J. Douglas Birdwell, Major Professor

We have read this thesis
and recommend its acceptance:

Tsewei Wang

Seddik M. Djouadi

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Exploration of Neural Structures for Dynamic System Control

A Thesis Presented for
The Master of Science
Degree

The University of Tennessee, Knoxville

Scott Frederick Hansen

December 2012

© by Scott Frederick Hansen, 2012
All Rights Reserved.

Abstract

Biological neural systems are powerful mechanisms for controlling biological systems. While the complexity of biological neural networks makes exact simulation intractable, several key aspects lend themselves to implementation on computational systems.

This thesis constructs a discrete event neural network simulation that implements aspects of biological neural networks. A combined genetic programming/simulated annealing approach is utilized to design network structures that function as regulators for continuous time dynamic systems in the presence of process noise when simulated using a discrete event neural simulation.

Methods of constructing such networks are analyzed including examination of the final network structure and the algorithm used to construct the networks. The parameters of the network simulation are also analyzed, as well as the interface between the network and the dynamic system. This analysis provides insight to the construction of networks for more complicated control applications.

Contents

List of Tables	vii
List of Figures	ix
1 Introduction	1
1.1 Problem Statement	2
2 Background	3
2.1 Neuro-physiology	4
2.1.1 Neuron Models	6
2.1.2 Networks of Neurons	9
2.1.3 Central Pattern Generators	10
2.1.4 Synaptic Plasticity	12
2.2 Pulse Frequency Modulation	13
2.2.1 Demodulation	14
2.3 Limited Biological Feasibility of Traditional Artificial Neural Networks	15
2.4 Evolutionary Programming	17
2.4.1 Genetic Programming	18
2.4.2 Simulated Annealing	19
2.5 Harmonic Oscillator	19
2.6 Inverted Pendulum	20
2.6.1 Pendulum Stabilization	21

2.6.2	Swing-up Control	23
2.6.3	Inverted Pendulum Models	23
2.7	Summary	24
3	Neural Simulation and Evolution	25
3.1	Discrete Event Neural Simulation	26
3.2	Single Neuron Behavior	27
3.3	Neural Evolution Algorithm	29
3.4	Network Visualization	35
3.5	Summary	37
4	Harmonic Oscillator	38
4.1	Unforced System	40
4.2	Network Optimization Examples	41
4.3	Neural Evolution Algorithm Statistics	50
4.4	Summary	55
5	Control of the Cart-and-Pendulum	56
5.1	Model	56
5.2	Cart-and-Pendulum Control Strategy	58
5.3	Improved Performance Using the Neural Simulator	61
5.4	Sumamry	69
6	Conclusions	70
6.1	Future Work	71
	Bibliography	72
A	Source Code	82
A.1	NeuralNetwork.h	82
A.2	NeuralNetwork.cpp	85

A.3	NeuralNetworkFactory.h	98
A.4	NeuralNetworkFactory.cpp	99
A.5	Breeder.h	107
A.6	Breeder.cpp	109
A.7	sim_harmonic.h	119
A.8	sim_pendulum_ctrl.h	127
A.9	mainmpi2.cpp	138
A.10	pendulum_ctrl.cpp	148
A.11	makefile	158
	Vita	160

List of Tables

3.1	A description of variables used in the master process.	32
3.2	A description of variables used in the slave processes.	34
3.3	Parameter details for the network visualization example.	36
3.4	Connection details for the network described in Table 3.3.	37
4.1	Details of the feedback network for the harmonic oscillator optimized without noise.	43
4.2	Connection details for the network described in Table 4.1.	43
4.3	Details of the feedback network for the harmonic oscillator optimized in the presence of a small amount of process noise.	47
4.4	Connection details for the network described in Table 4.3.	47
4.5	Average performance of Network 2 over 50 simulations shows favorable performance even in the presence of process noise.	50
5.1	Parameters used for the cart-and-pendulum system.	58
5.2	Parameter values for the network in Figure 5.4.	64
5.3	Connection values for the network in Figure 5.4.	65
5.4	Parameter values for the network in Figure 5.5.	66
5.5	Connection values for the network in Figure 5.5.	66
5.6	Parameter values for the network in Figure 5.6.	67
5.7	Connection values for the network in Figure 5.6.	68
5.8	Parameter values for the network in Figure 5.7.	69

5.9	Connection values for the network in Figure 5.7.	69
-----	--	----

List of Figures

2.1	A simple diagram of a biological neuron [32].	6
2.2	The inverted pendulum.	21
3.1	Single neuron with staircase input.	28
3.2	Single neuron with sinusoidal input.	29
3.3	A sample network observed through the network visualization utility.	36
4.1	Block diagram of the harmonic oscillator with discrete event network feedback.	39
4.2	Examples of band-limited noise processes with differ filter parameters.	40
4.3	Behavior of the unforced harmonic oscillator.	41
4.4	Network construction in the absence of process noise results in simple network structures.	42
4.5	Behavior of the closed loop harmonic oscillator without noise using a feedback network developed in the absence of process noise	43
4.6	Behavior of the closed loop harmonic oscillator with a small amount of process noise using a feedback network developed in the absences of process noise.	44
4.7	Network developed in the presence of process noise.	46
4.8	Harmonic oscillator with feedback using a network developed with a small amount of process noise.	48

4.9	Performance of the harmonic oscillator with process noise with $\alpha =$ 0.01 using a feedback network developed in the presence of process noise.	49
4.10	Performance of the harmonic oscillator with process noise with $\alpha = 1.0$ using a feedback network developed in the presence of process noise. .	50
4.11	Average execution time and average generations required to find a network solution decrease as the population size increases.	51
4.12	The average number of neurons explored by the neural evolution algorithm is reduced on average when the population size is increased, resulting in less complex network structures.	53
4.13	The average number of synapses explored by the neural evolution algorithm is reduced on average when the population size is increased, resulting in less complex network structures.	54
5.1	The inverted pendulum on a cart.	58
5.2	Block diagram of the closed loop system including a neural network simulation that improves the performance of swing-up control of the inverted pendulum.	61
5.3	An appropriately tuned network improves the robustness of the closed loop system.	63
5.4	A simple network structure developed when the neural simulation algorithm proceeds in the absence of process noise.	64
5.5	Neural network structures for neural evolution for process noise with a gain of 1.0	65
5.6	Neural network structures for neural evolution for process noise with a gain of 1.5	67
5.7	A more complex neural network structure developed for neural evolu- tion in the presences of process noise with a gain of 2.0.	68

Chapter 1

Introduction

Biological neural systems function as powerful feedback controllers that regulate a wide variety of biological processes. These systems implement memory formation, intelligent decision making, and intricate parallel regulatory activities. The complexity of biological neural systems requires that simplifications be made to capture the behavior of systems in simulation.

This thesis is an exploratory study of a method for modeling neural systems using a discrete event neural simulation that emulates aspects of biological neural networks. The network simulation emphasizes the manner in which voltage pulses are transmitted in biological neural systems. These pulses are treated as discrete events, avoiding computationally expensive numerical methods traditionally used to evaluate neural behavior described by sets of coupled nonlinear differential equations. The discrete event neural simulation provides a better representation of biological systems than traditional artificial neural networks, which typically do not capture network complexity, communication delays, or pulse modulation.

The behavior of the discrete event neural simulation is explored in a hybrid systems setting formed by interfacing the simulation with two different continuous time dynamic systems: the harmonic oscillator and the inverted pendulum. The harmonic oscillator is a simple problem that provides an initial benchmark of the

neural simulation. The inverted pendulum is a more complex problem commonly used as a benchmark for control systems. For each application, distinct network structures are developed using a neural evolution algorithm that combines a genetic algorithm and simulated annealing.

Experiments with the neural simulation result in closed loop systems that improve performance, measured by weighted least squares criteria appropriate to each problem, in the presence of process noise. Network structures are analyzed, and important simulation parameters indicated. Knowledge gained from these experiments provides insight to the construction of discrete event networks and how more effective neural structures might be developed in future applications.

Chapter 2 describes biological neural systems as well as tools utilized in the construction and implementation of the discrete event neural simulation. Chapter 3 presents the discrete event neural simulation model. Chapter 4 evaluates the construction and performance of discrete event neural networks utilized as feedback controllers for the harmonic oscillator. Chapter 5 applies the neural simulation as a regulator for the cart-and-pendulum. Chapter 6 concludes with a discussion of the findings and possible directions for future work. We begin with a concise statement of the problem that motivates this work.

1.1 Problem Statement

Explore the behavior of a discrete event neural simulation with similar behavior to biological neural systems, and observe hybrid system interactions when implemented as a regulator for continuous time dynamic systems. Construct and analyze network structures used by the simulation that provide improved performance for these systems.

Chapter 2

Background

This chapter covers useful information applicable to the construction and application of discrete event neural simulation. The discrete event neural simulation implements several aspects of biological neural systems. Hence, an understanding of neurophysiology is helpful in justifying the structure of the neural simulator. These biological systems are governed by complex electro-chemical processes. Several key properties, such as charge accumulation, transmission of pulses, and transmission delay can be efficiently simulated in a computational environment in the form of a discrete event neural simulation.

Given a model of neural behavior and a method of simulation, one must determine how to interface the discrete event system to another system. In this research the simulation interacts with systems with continuous time and continuous state dynamics. Pulse frequency modulation (PFM) is a common modulation method used in communication systems and describes the behavior of a single neuron to a limited extent. Methods for demodulated PFM are used to build interfaces from the discrete event neural simulation to a system with continuous time dynamics. The interface from the continuous system to the discrete event system can be implemented using integrate-and-fire neuron models with a threshold and defined refractory periods to limit firing rates.

Construction of the discrete event network is accomplished by an evolutionary algorithm approach that is similar to methods used to construct some traditional artificial neural networks. Traditional artificial neural networks are a common computational tool, loosely inspired by biological neural systems. These networks have a plethora of applications, but fall short in accurate representation of biological systems. One method of construction of traditional artificial neural networks is the use of evolutionary programming.

Two systems with continuous time dynamics are used to explore the behavior of the discrete event neural simulation. The harmonic oscillator is a two state system that oscillates freely without friction or external input. This is a system with simple dynamics. The cart-and-pendulum is a nonlinear system commonly used as a benchmark for control systems. The cart-and pendulum represents a more challenging problem since multiple states must be controlled from a single input where significant nonlinearities and coupling exist. For both systems, control is achieved by driving the states of the system in a manner that keeps them near a desired equilibrium value.

2.1 Neuro-physiology

Biological neural systems are composed of cells called neurons. A typical neuron (Figure 2.1) consists of a soma, axon, and dendrites. Each axon terminal of a neuron typically connects to dendrites of other neurons through a gap called a synapse. Other connections are possible, such as an axon connecting to the soma of another neuron; this, however, is less common. Self-connections can exist where the axon of a neuron connects to its own dendritic tree or soma. The neuron contains concentrations of charged particles, or ions (e.g. Na^+ , K^+ , Ca^{2+}), which move in response to concentration gradients and electric fields. Ions can cross cell membranes via ion channels, which open and close in response to the voltage potential across the membrane and ion pumps, which provide an active transport mechanism. Oppositely

charged ions attract and therefore congregate on opposite sides of the membrane and produce an electric voltage potential called an action potential. When the action potential of a neuron reaches a threshold, channels open at the axon terminal of a neuron allowing particles called neuro-transmitters to travel across the synaptic cleft to the dendrite of another neuron, where they are received by the post-synaptic neuron, causing changes in ion flow and the neuron membranes voltage potential. The voltage transient induced by ion flow is referred to as an action potential [36].

These action potentials initiate pulse events transmitted to other neurons. When a neuron fires an action potential, there is period in which the neuron is not physiologically capable of firing another pulse. This period is referred to as the refractory period of a neuron and limits the maximum firing rate of the neuron.

The penultimate example of an effective biological neural network is the human brain. The human brain has on average 85 billion neurons with an average of 10^{12} synaptic connections [27]. Regions of the brain are highly specialized and perform specific tasks. The behaviors of these regions are governed by the density of neurons, neuron types, the density of synapses, the shape of a neuron's dendritic trees, neuron membrane surface morphology, contact location of excitatory and inhibitory synapses, synaptic plasticity, inter-synaptic distance, mylenation, axon length, and many other factors [58]. This results in an extraordinarily complex and chaotic system, albeit one capable of intricate decision making, memory and regulation.

In part, the effectiveness of biological neural networks is due to their adaptability. The morphology of the dendritic tree of a neuron can change. These changes typically occur often in developing neural systems, but still occur in mature neural systems in response to changing sensory input and experience [15, 41]. The efficacy of a synaptic connection between an axon and dendrite can be enhanced through a complex biochemical process called long-term potentiation (LTP, Section 2.1.4). An analogous process called long-term depression (LTD) can decrease the efficacy of a connection. LTP and LTD are mechanisms of synaptic plasticity, which is a driving force behind cognition and learning [26, 51]. Mylenation can also enhance the performance of

biological neural networks by increasing the speed of signal transmission in an axon. Mylenation is derived from a specialized lipid based membrane that surrounds the axon of certain neurons [35]. The development of mylenation has been show to be adaptable to an organisms environmental needs and experiences [20].

Despite the variety of neurons and their connections, biological neural networks can be abstracted to a few simple concepts that can be applied to computational systems. Neurons are essentially accumulators that exhibit threshold stimulated events [24] and are connected by communication pathways (axons, synapses, and dendrites) along which signals (action potentials) propagate. The behavior of a neural network is influenced by the characteristics of individual neurons such as firing thresholds, refractory period and morphology, by the network structure connecting the neurons, and by the signal delays along the pathways.

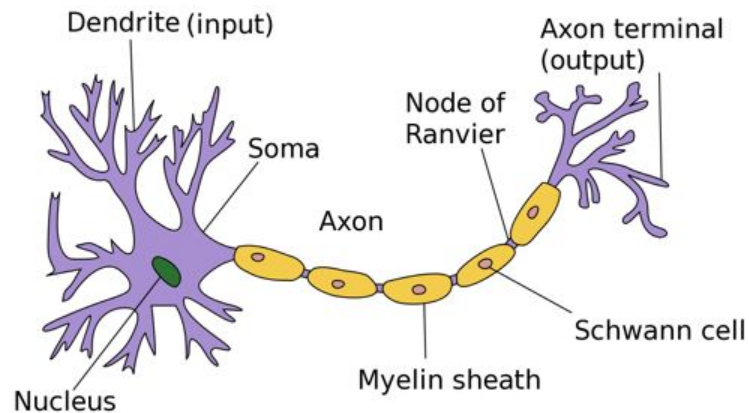


Figure 2.1: A simple diagram of a biological neuron [32].

2.1.1 Neuron Models

There are three general types of neurons: sensory neurons, motor-neurons, and inter-neurons. Sensory neurons translate information gathered from the environment to action potentials. Motor-neurons perform the complementary operation, mapping action potentials to stimuli for muscle activity. Inter-neurons form the complex network that connects sensory neurons and motor-neurons.

Most neurons form relatively sparse connections primarily to nearby neurons through synapses; other types of neurons form long distance connections such as are found in the spinal chord or in the white matter connecting functional regions of the brain [58]. Synapses are either excitatory or inhibitory based on the type of neurotransmitter released by the axon. Excitatory synapses stimulate activity by increasing nerve membrane potentials in post-synaptic neurons. Inhibitory synapses decrease activity by decreasing membrane potentials. [36]

Neurons implement an integrate-and-fire mechanism. The simplest neural model describes current flowing through a neuron by

$$C_m \frac{dV_m}{dt} = I(t) - \frac{V_m}{R_m} \quad (2.1)$$

where C_m is the cell capacitance, V_m is the membrane voltage, R_m is a leakage resistance and I is the input current [24]. This model treats the neuron as a simple integrator. The input current models the effects of neurotransmitters released by pre-synaptic neurons, and these currents affect learning and adaptation (e.g. LTP/LTD, myelination). This model can be further developed by including a refractory period and a partial threshold reset of the integrator value.

More accurate descriptions of the shape of action potential events are described by differential equations relating the currents generated by ions moving through the axon and across the synapse to voltage changes in the cell membrane. The four state Hodgkin-Huxley (HH) equations are the canonical model for describing the current flow through nerve membranes during action potential events and represent an historic breakthrough in modeling neurological processes [28]. A general HH model is given

by the equations

$$C\dot{v}(t) = g_L(v_L - v(t)) + gN_a m^3(t)h(t)(vN_a - v(t)) + Kn^4(t)v_K - v(t) + I \quad (2.2)$$

$$\dot{m}(t) = \frac{m_\infty(v(t)) - m(t)}{\tau_m(v(t))} \quad (2.3)$$

$$\dot{h}(t) = \frac{h_\infty(v(t)) - h(t)}{\tau_h(v(t))} \quad (2.4)$$

$$\dot{n}(t) = \frac{n_\infty(v(t)) - n(t)}{\tau_n(v(t))} \quad (2.5)$$

where $v(t)$ is the membrane potential, $m(t), h(t), n(t)$ describe activation of ion channels, $m_\infty, h_\infty, n_\infty$ are nonlinear voltage dependent steady state conductances, and I is the input current [51]. For a description of the other parameters, see [51]. While this model accurately describes the shape of action potentials in a single neuron when compared to experimental data, it is not appropriate for an arbitrarily complex network composed of a large number of neurons.

A simplification of the HH model is the two state Bonhoeffer-van der Pol (BVP) model [21], often referred to as the FitzHugh–Nagumo model:

$$\dot{x} = \mu x - cx^3 - y + I \quad (2.6)$$

$$\dot{y} = x + by - a \quad (2.7)$$

where x is the membrane voltage potential, y is the fast current dynamics, and I is an input current [51]. This model is often considered due to its simple equivalent circuit and ease of interpretation [45]. The parameters a, b, c determine the shape of the pulse and delay between pulses. Under certain parameter conditions, the BVP model describes a stable van der Pol oscillator which exhibits limit cycle behavior with appropriately chosen parameters. The pulses generated by this model also correspond closely to experimental data, but for the same reason as with the HH model, simulation does not scale to large networks.

While these models (and others) can accurately describe the shape of action potentials, they require numerical integration to evaluate. This is useful for studying the interaction of a few neurons, but quickly becomes a computationally challenging task when many neurons are coupled together in a complex network. It is not clear that highly detailed neuron models are either necessary or desirable when modeling and simulating a large network. The presence and timing of pulses, rather than their shapes, is the most important attribute. The discrete event neural simulator avoids the computationally expensive task of evaluating large collections of coupled differential equations and focuses on the transmission and timing of the pulse events.

2.1.2 Networks of Neurons

While the behavior of an individual neuron plays a significant role in determining the activity of a neural network, the connectivity of the neurons within the network heavily influences the behavior of the network. Networks of neurons typically have cyclic connections that result in neuron firing events that repeat [12]. Since neurons are distributed spatially, this periodic behavior is typically referred to as a spatiotemporal oscillation or reverberation [51]. A pre-synaptic neuron fires a pulse, which can trigger events in some or all of the post-synaptic neurons. Those neurons fire, causing post-synaptic neurons to fire, which triggers additional firing events. The firing events can be tracked from one neuron, through a chain of neurons, back to the original neuron. This is referred to as a spatial oscillation since the neurons have a defined spatial location and a pulse can be tracked through this space. A single neuron fires repetitively. This is referred to as a temporal oscillation. In this manner, firing events can be traced in both time and space. Synchronization of spatiotemporal oscillations between coupled sub-networks of neurons plays a crucial role in memory formation, perception, consciousness, and muscle control [12, 40]. This synchronization effect was postulated by Wiener from his observation of the frequency spectrum of measured electrical signals from the brain [64].

Both propagation delays along axons and dynamics induced by the integrate-and-fire behavior of the neurons play significant roles in the determination of spatiotemporal oscillatory behaviors. Limit cycle behavior in small networks modulates limit cycle behavior in larger neural networks, and small networks tend to have a higher oscillation frequency than large networks [12]. Both transmission delay and the number of intervening neurons increase with distance along pathways between neurons. These factors influence the dynamic behavior of the neural network and tend to reduce spatiotemporal oscillation frequency.

The existence of spatiotemporal oscillations is partially dependent on the delay in transmission of action potentials between neurons. The units of charge in neural currents are ions, commonly Na^+ , K^+ , and Ca^{2+} , which have large mass and, consequently, a slow velocity relative to electron current flow. Conduction velocity can range from 1–120 $\frac{m}{s}$ [36], depending on myelination. The time duration of action potentials due to ion flow takes on the order of tens of milliseconds [36]. Propagation time is dependent on the axon length and presence of myelination. The delay in the communication between neurons can have a nontrivial influence on oscillatory behaviors due to either length (for long myelinated axons) or the number of intervening neurons.

2.1.3 Central Pattern Generators

The discrete event neural simulator consists of a network of simulated neurons; hence, it is useful to understand how biological networks can be analyzed. Due to the overwhelming complexity of large scale neural systems, analysis of biological neural networks is often limited to small structures. One class of structures often examined is the central pattern generator (CPG). A CPG is a neural microcircuit that controls motor activity or other periodic biological functions through neural networks that exhibit oscillatory behavior in the absence of sensory feedback [51]. Central pattern generators are a well-established field of study and have been observed in biological

systems [14], simulated on computer systems [7, 11, 18, 37], and implemented on robotic platforms [30, 31].

One advantage of using CPG's in simulation studies is the relative simplicity of their underlying network structure. When grown on a substrate with a fixed structure, sustained reverberations have been observed in networks of rat hippocampal neurons with as few as 20 to 100 neurons [40, 62]. This allows for the development of useful mathematical models and computer simulations of CPG behavior [14, 37, 61, 18, 11].

The effects of CPG's can also be observed in biological organisms with simple neurological systems. For example, when a lamprey's spinal chord is removed and immersed in a bath of excitatory amino acids, the amino acids stimulate the spinal chord, and the neural behavior can be measured [14]. Stimulation of this biological CPG generates oscillatory activity in the spinal chord responsible for the lamprey's swimming behavior, an inherently oscillatory activity. Firing patterns of neuronal groups in the mollusk have also been measured [37]. This allows for analysis of recurrent biological CPG's that control the mollusk swimming and hunting behaviors [61]. The behavior of CPG models and simulations can be compared to the behavior of CPG's in biological organisms *in vivo*. This approach, however, is generally limited to vertebrae and invertebrates with simple neural systems.

The modeling and simulation of CPG's takes place on two levels. From a high level perspective, CPG's are treated as black box oscillators where multiple CPG's are coupled together in a particular manner to achieve a desired behavior [14, 30, 31]. The simplest models treat each CPG as a sinusoidal oscillator. Due to the simplicity of implementing these CPG's, this approach is often utilized in the control of robots whose movements mimic the behavior of biological organisms that exhibit oscillatory movement, such as the lamprey, salamander, or snake [30, 31].

Detailed analysis of the individual central pattern generator examines attributes such as the types of neurons found in the CPG, the connections between the neurons within the CPG, and the connections between coupled CPG's. A typical CPG

consists of an arrangement of sensory neurons, motor-neurons, and interneurons with inhibitory and excitatory connections [11, 18].

Central pattern generators represent a type of associative memory container [37]. A pattern is embedded in the oscillatory firing behavior of the network. The pattern is observed in the behavior of the motor neurons and accessed by exciting sensory neurons. The power of this interpretation lies in the fact that multiple patterns of arbitrary length can potentially be store in a CPG and accessed by stimulating the network in different ways.

2.1.4 Synaptic Plasticity

Reinforcing the connection between neurons that exhibit a high degree of correlated activity is a classical concept in neuro-systems theory [26]. The synapses between neurons are subject to prolonged dynamic changes in efficacy through complex biochemical mechanisms referred to as long term potentiation (LTP) and long term depression (LTD) [19]. Dynamic synaptic plasticity is an underling factor in several neural processes including learning and memory formation in the brain [9]. LTP reinforces apparently causal neural connections; LTD reduces the strength of non-causal or uncorrelated interactions.

Consider a neural connection which is composed of a presynaptic neuron, a post synaptic neuron, and a synapse adjoining the two neurons. If the presynaptic neuron fires at time t_{pre} and the post-synaptic neuron fires at time t_{post} , let $\Delta t = t_{post} - t_{pre}$ be the difference in firing times. If there exists a small $\varepsilon > 0$ and $\Delta t \in (0, \varepsilon]$, then this indicates the possibility of a causal relationship between the firing events. In this case, the LTP mechanism strengthens the synapse between the two neurons. If $\Delta t \in [-\varepsilon, 0)$, this indicates that the firing of the post-synaptic neuron is not likely dependent on the firing of the post-synaptic neuron. [51] The biological mechanism by which a synapse is strengthened or weakened exhibits an exponential ($ae^{-\frac{\Delta t}{b}}$) characteristic, where a and b are positive constants for LTP and negative for LTD [8].

Thus if the $\Delta t > 0$ is small, LTP increases in synaptic efficacy or coupling between the two neurons. If $\Delta t < 0$ has a small magnitude, LTD decreases the synaptic efficacy or coupling.

2.2 Pulse Frequency Modulation

The discrete event neuron used by the discrete event neural simulation functions in a similar manner to a technique from communication systems called pulse frequency modulation (PFM). The behaviors of biological neurons can be described in this manner, including the behavior of a single integrate-and-fire neuron [53, 6]. The integral pulse-frequency modulated (IPFM) signal $y(t)$ for an input signal $x(t)$ is given by the equations

$$\frac{dp(t)}{dt} = x(t) - r \operatorname{sgn}(p(t))\delta(|p(t)| - r) \quad (2.8)$$

$$y(t) = \operatorname{sgn}(p(t))\delta(|p(t)| - r) \quad (2.9)$$

where p is the value of the integrator, r is a threshold, and sgn is the sign function [49]. The function δ is a unit impulse where

$$\delta(a) = \begin{cases} 1 & \text{if } a = 0 \\ 0 & \text{otherwise} \end{cases}$$

When the value of the integrator exceeds the threshold r , a pulse occurs at that time and the value of the integrator is reset.

Modulating a signal by IPFM results in a series of pulses whose frequency of occurrence increases as the magnitude of the input signal increases. Note that the value of the integrator increases for positive x and decreases for negative x . If the magnitude of the input signal is not large enough to exceed the threshold, there will be no non zero output, even though x contains non zero values.

While IPFM describes the characteristic behavior of a single neuron, the input signal $x(t)$ is a continuous time signal and cannot convey pulse events transmitted from other neurons in a network of IPFM neurons. Modification of the traditional IPFM model can be made to take neuron interactions into account. Suppose there are N neurons with threshold r_i and accumulator values p_i for $i = 1, \dots, N$. If these neurons are connected to a neuron described by IPFM, with connection weights $w_1, \dots, w_N$, then the dynamic accumulator equation can be adapted such that

$$\frac{dp(t)}{dt} = x(t) - r \operatorname{sgn}(p(t)) \delta(|p(t)| - r) + \sum_{i=1}^N w_i \delta(|p_i(t)| - r_i) \quad (2.10)$$

This model takes into account pulse events transmitted by other neurons, but does not model the effect of refractory period which is better described algorithmically. In this case, the signal x is external to the network.

2.2.1 Demodulation

Since a neuron implements IPFM, demodulation of a neuron's output is necessary to provide an interface from a neural network to a physical system. In order to use an integral pulse-frequency modulated signal as a control signal, it is necessary to demodulate the signal. Determining the instantaneous frequency at which pulses occur in the modulated signal is sufficient for demodulation, but cannot be exactly implemented. In practice, one updates the demodulated signal when pulses occur in the modulated signal using an estimation scheme such as a low pass filter to interpolate between measurements [2].

An efficient way to low pass filter an IPFM signal is the exponentially weighted moving average (EWMA). The EWMA is a digital low pass filter derived from applying the backwards difference method to a first order continuous time low pass filter. For a time constant τ , the transfer function for a first order low pass filter between a continuous time input signal $X(s)$ and continuous time output $Y(s)$ is

given by

$$H(s) = \frac{1}{1 + \tau s} = \frac{Y(s)}{X(s)} \quad (2.11)$$

Using the inverse Laplace transform and sampling at rate T_s , a sampled system is produced,

$$\tau \dot{y}(nT_s) + y(nT_s) = x(nT_s) \quad (2.12)$$

where the derivative can be approximated by the backwards difference such that

$$\dot{y}(nT_s) = \frac{y(nT_s) - y((n-1)T_s)}{T_s} \quad (2.13)$$

Let $x[n] = x(nT_s)$ and $y[n] = y(nT_s)$. Using this approximation, the discrete time system is

$$y[n] = (1 - \alpha)y[n-1] + \alpha x[n] \quad (2.14)$$

where $\alpha = \frac{T_s}{T_s + \tau}$.

The EWMA low pass filter is a computationally efficient implementation of a first order digital low pass filter since it can be computed recursively [33]. The behavior of the EWMA filter is governed by a single parameter $\alpha \in (0, 1]$. Note that the corner frequency w_c of the low pass filter is $w_c = \frac{1}{\tau}$ for the time constant τ . Hence, the EWMA filter parameter α is directly related to the corner frequency of the filter. A small value of α corresponds to a low corner frequency.

2.3 Limited Biological Feasibility of Traditional Artificial Neural Networks

The discrete event neural network simulation is considerably different than traditional artificial neural networks. Traditional artificial neural networks (TANN) are a common computational tool that map an input space to an output space through a series of three or more layered sets of artificial neurons. The input layer first accepts a

pattern defined by inputs to each neuron of the layer, then feeds that pattern through one or more hidden layers, and finally provides a classification of that pattern at the output layer, defined by an output from each neuron of the output layer. Let x_i for $i = 1, \dots, N$ be the numerical values of the states of neurons in a layer, and let y_j be the j^{th} neuron in the next layer. Let $n_j = \sum_{i=1}^N w_{ij}x_i$, where w_{ij} is the weight from x_i to y_j . Then the value of the state y_j is

$$y_j = f(n_j) \tag{2.15}$$

where the function f is a threshold activation function. Feedback, generally in the form of a gradient descent method such as back propagation, alters weights between the neurons in each layer to minimize the error between the output layer and desired output [17]. A gradient descent method requires a smooth objective function; hence the activation function f is usually a sigmoid function instead of a discontinuous Heavyside step function.

The advantage of such a computational system is its generality. The TANN can handle mappings of input to output spaces that have unknown linear or nonlinear functions. That is, tuning of the TANN can take place without any knowledge of the underlying system other than the example input and output patterns used for training [4]. There are limits to the utility of ANN's backpropagation, which is a gradient descent method and hence prone to achieving local optimization instead of global optimization. Stochastic optimization algorithms and evolutionary programming are often used in conjunction with backpropagation to search multiple optima [55, 56, 57, 60].

Dynamic behaviors can also be modeled by adding integrators to the network. In this case, for a dynamic system modeled in state space form $\dot{\underline{z}} = \underline{f}(\underline{z}, \underline{u})$ with output $\underline{w} = \underline{g}(\underline{z}, \underline{v})$, the Tonne's are used to represent the functions $\underline{f}$ and $\underline{g}$. The Tawny's are typically trained using recorded sampled time series data. The neural

network constructs a model for the system and can be used for system identification and control [46, 39, 48].

The space of network structures and weights in traditional artificial neural networks is large and difficult to search efficiently. Topology and weight evolving artificial neural networks (TWEANN) reformulate the traditional backpropagation approach to tuning traditional artificial neural networks by evolutionary programming and genetic algorithms (Section 2.4) to search the space of network structures and weights. Two related approaches to building TWEANN's are Neuro Evolution of Augmented Topologies (NEAT) [60] and Modular Neuro Evolution of Augmented Topologies (Modular NEAT) [55]. Both methods keep track of specific network structures. NEAT implements speciation, which allows evolution of a single network structure before permitting competition of that network with other networks in a population of possible solutions. Modular NEAT extends this concept by encoding sub-networks that can be combined to construct more complex network structures. These methods improve the performance of optimizing artificial neural network over the use of backpropagation alone.

Traditional artificial neural networks have little in common with biological neural systems. Biological networks have highly cyclical connections that are sparsely connected. TANN's typically have only feed forward connections that are densely connected between layers. Cyclical connections can be implemented in TANN's; however, in this case backpropagation cannot be used. TANN's do not provide for pulse modulation or transmission delay, which are important characteristics of biological neural networks. A model for these properties is used in the discrete event neural simulation.

2.4 Evolutionary Programming

Biological neural systems are a result of inter-generational adaptations and random mutations that have taken place over a significant period of time. Evolutionary

programming emulates the biological evolutionary process to solve an optimization problem. Evolutionary programming generates a population of solutions, evaluates their performance, and uses the top performers as a basis for stochastically generating a new population of possible solutions. [17] A neural evolution algorithm is used to develop network structures to be utilized by the discrete event neural simulator. Evolution of neural networks can refer to dynamic changes in the properties describing neurons (learning rules), simple changes in the connection weights between neurons, and global reconfigurations of network structure [66].

Stochastic optimization algorithms are advantageous when performing optimization in systems where gradient descent methods are impractical. Such systems can have discontinuous objective functions where the gradient is ill-defined, or where the objective function itself is not well-defined. They are also useful when dealing with functions that have a large number of local optima and gradient descent methods converge to local optima instead of global optima. Stochastic techniques tend to have slower convergence times but can explore more of the solution space.

2.4.1 Genetic Programming

Genetic programming is a flavor of evolutionary programming that can be used to optimize neural networks [17]. A potential solution to an optimization problem is encoded in a ‘chromosome’, typically a binary or character string. The chromosomes are ranked based on a performance metric. The top performing chromosomes then undergo a process called ‘crossover’. Crossover exchanges segments of two chromosomes to generate two new chromosomes. The process is repeated with the new chromosomes included in the population, after possible random mutations. This process is analogous to biological processes.

One challenge in applying genetic algorithms to neural networks is network encoding. The structure, weights, and other network properties must be encoded in the chromosome that the algorithm uses in crossover. While weights do not present a

problem, encoding a network structure that functions across multiple generations does. Several such encoding schemes exist [38, 60, 55]. The neural evolution algorithm used here does not implement crossover because of the difficulties of network encoding. Inter-generational advances are performed through random mutation.

2.4.2 Simulated Annealing

Another optimization method used by the neural evolution algorithm is simulated annealing. Simulated annealing is a stochastic optimization algorithm that is useful in the search for global optima for continuous and discontinuous problems [16]. The algorithm randomly perturbs a point in the solution space, possibly by a large amount. If the new point performs better, it is accepted as a better solution. Perturbations that do not perform better are accepted with decreasing probability as a function of the number of completed iterations. This allows the algorithm to explore larger areas of the solution space without becoming trapped in local optima. Neither genetic programming nor simulated annealing, however, guarantees convergence to a locally or globally optimal solution in finite time.

2.5 Harmonic Oscillator

Initial exploration of the discrete event neural simulation observes the hybrid interaction of the simulation with the continuous time harmonic oscillator. The harmonic oscillator is a dynamic system described by a second order differential equation.

$$m\ddot{y}(t) + ky(t) = u(t) \tag{2.16}$$

In a physical system, such as a frictionless mass-spring, m is a mass, y is a position, k is constant where $\frac{1}{2\pi}\sqrt{\frac{k}{m}}$ is the frequency of oscillation, and u is a forcing function. If $u(t) = 0$ and the initial condition is non zero, the trajectory of the system will oscillate sinusoidally. Stabilization of the harmonic oscillator is accomplished by

driving the trajectory to zero, which can be accomplished by a well-timed impulse input of sufficient magnitude. This system presents an interesting, albeit simple, system to use as a benchmark.

2.6 Inverted Pendulum

The stabilization of an inverted pendulum (Figure 2.2) is a canonical control system problem with nonlinear dynamics that is useful for benchmarking linear and nonlinear control algorithms, and is used as for exploration of the discrete event neural simulator. The unforced pendulum is a nonlinear system that can be described by a second order differential equation derived from Newton’s second law of motion [34]

$$ml\ddot{\theta} = -mg\sin\theta - kl\dot{\theta} \quad (2.17)$$

where m is a point mass located at a distance l from the axis of rotation, g is the gravitational constant, and k is a friction coefficient. The pendulum has similar behavior to the harmonic oscillator, but has two equilibrium points, one of which is unstable. The behavior of the pendulum is governed by nonlinear dynamics. The inverted pendulum has a simple physical interpretation: gravitational force pulls the pendulum downward to a stable equilibrium. This system describes a variety of electro-mechanical systems with nonlinear second order dynamics (e.g. synchronous generators, Josephson junction circuits, phase-locked loops) [34].

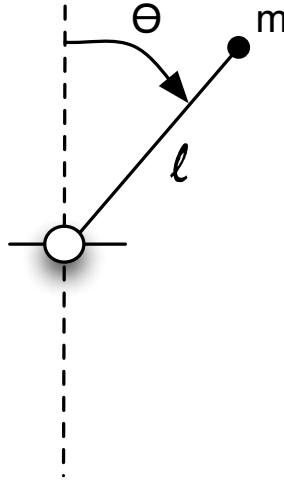


Figure 2.2: The inverted pendulum.

The objective of the inverted pendulum control problem is to maintain the angular position of the pendulum in the vertical upright position, an unstable equilibrium position where $\theta = 0$. Gravitational force tends to pull the pendulum to the downward vertical position, which is a stable equilibrium, at $\theta = \pi/2$. Control is accomplished by moving the base of the pendulum, which also has mass. There is only one input to the system controlling the acceleration of the base. This single input is used to control the multiple states associated with the system. Many models exist describing the interaction between the base and the pendulum as well as the behavior of the pendulum itself, generating control problems of varying degrees of complexity. Two problems formulated for the inverted pendulum are stabilization of the pendulum and swing-up control of the pendulum.

2.6.1 Pendulum Stabilization

The simplest inverted pendulum problem involves stabilizing the pendulum given a small angular displacement from the unstable equilibrium. This is a simple problem commonly solved by linearizing the dynamics of the pendulum around the unstable

equilibrium point and applying one of a variety of linear control techniques (e.g. linear quadratic regulator [5]) to provide control. Robust control methods can also be used, typically based on H_∞ design [50]. Traditional artificial neural networks are also often used to solve this problem [4, 56, 57].

A linear quadratic regulator provides optimal control for a linear system [5]. Since the pendulum is a nonlinear systems, the state space model must be linearized near the unstable equilibrium point to create a linear approximation of the system

$$\dot{\underline{x}}(t) = A\underline{x}(t) + B\underline{u}(t) \quad (2.18)$$

where $\underline{x}(t)$ is the state of the systems and $\underline{u}(t)$ is the control input. The LQR problem finds a control input $\underline{u}$ that minimizes a quadratic cost function

$$J(\underline{u}) = \int_0^\infty (\underline{z}^T(t)Q\underline{z}(t) + \underline{u}(t)^T R\underline{u}(t)) dt \quad (2.19)$$

where $\underline{z} = M\underline{x}$ denotes the regulated variables and Q and R are real symmetric positive semidefinite weighting matrices. This results in an optimal control law

$$\underline{u}(t) = K\underline{x}(t) = -R^{-1}B^T P\underline{x}(t) \quad (2.20)$$

where P is the solution to the algebraic Riccati equation $A^T P + PA - PBR^{-1}B^T P + M^T Q M = 0$, which exists and is unique if $(A, B, Q^{1/2}M)$ is stabilizeable and detectable.

Limitations of this approach are related to the linearization process. Control is restricted to a small displacement from the equilibrium point where the system is linearized.

2.6.2 Swing-up Control

A more challenging problem is to stabilize the pendulum at the unstable equilibrium point starting from the stable equilibrium. This is referred to as swing-up control. The challenge is due to the nonlinearities present in the system and the effect of gravity. There are a wide variety of methods that provided effective swing-up control of the inverted pendulum. These methods include methods include partial feedback linearization methods [10, 25, 59] and more exotic strategies like grey prediction modeling [29]. The prevalent method for swing-up control of the inverted pendulum uses energy based strategies [3, 54, 67]. The control law is derived from the rotational kinetic energy and potential energy of the pendulum, such that the energy of the system is driven to the desired value corresponding to the potential energy associated with the unstable equilibrium point. If a Lyapunov stability criteria is met, this is ensured to be a stable control method.

The control methods described here can be applied to a variety of pendulum models. There are several models commonly used in working with inverted pendulum systems. Implementation of a control strategy for an inverted pendulum requires detailed knowledge of a specific pendulum model as well as a model for the control input to the pendulum.

2.6.3 Inverted Pendulum Models

The inverted pendulum problem can be realized in a variety of ways, all of which involve balancing a pole in the vertical upright position. The canonical model is the cart-and-pendulum (Figure 5.1) [25, 29, 43, 67, 68], where the pendulum is fixed to a cart that can move in one or two directions. Generally the linear displacement of the cart is physically limited by the length of the track along which the cart moves. Another arrangement is the rotational pendulum (often referred to as the Furuta pendulum) where the base of the pendulum is attached to a horizontal arm that rotates. [22, 3, 52, 13, 23, 63, 65]. This has the advantage of a more compact physical

implementation where the motion of the base of the pendulum is limited only by the actuator controlling its motion and not by a physical boundary. Other formulations include the wheeled inverted pendulum [47].

Constructing inverted pendulum systems with more states to control adds complexity to the control problem. Two pendulums with different lengths and masses attached to the same base are referred to as the dual-inverted-pendulum [42, 50]. The pendulum itself may consist of multiple jointed segments [43, 65], which is also referred to as a dual-inverted-pendulum when there are two separate segments. These models, of course, extend to an arbitrary number of pendulums and joints. The addition of each pendulum or joint adds two coupled states to the system, hence increasing the number of states controlled by a single input. This project utilizes a cart-and-pendulum model. The model is discussed in detail in Chapter 5.

2.7 Summary

This chapter covers a variety of topics relevant to the development of a discrete event neural simulator. A discussion of properties of biological neural systems justifies their use in computational simulation. Construction of networks used by the discrete event neural simulator networks takes place by evolutionary programming techniques in the form of a neural evolution algorithm. Exploration of the neural simulator takes place by observing the interactions of discrete event neural networks with systems that have continuous time dynamics, in particular the harmonic oscillator and the inverted pendulum.

Chapter 3

Neural Simulation and Evolution

This chapter outlines methods used to develop a discrete event neural simulation. All calculations are performed using the C++ programming language. The neural network is a C++ class written by the author. The closed loop systems discussed in this work are hybrid systems consisting of a discrete event neural simulation coupled with a continuous time dynamic system. A neural evolution algorithm constructs network structures used by the simulator. This algorithm uses a distributed evolutionary programming approach combining a genetic algorithm with distributed simulated annealing.

The dynamics of the discrete event neural simulation are separate from the dynamics of the continuous time systems used to explore the behavior of the simulation. Numerical evaluation of the continuous time dynamic systems are performed with the *Livermore Solver for Ordinary Differential Equations (LSODE)* available from <http://www.netlib.org/odepack/>. Evaluation of the neural evolution algorithm takes place on distributed computational clusters. Two clusters are used. The first is provided by the Laboratory for Information Technologies consisting of 48 computational cores distributed across 10 nodes. This work also used the Newton High Performance Computing Cluster, a general purpose research cluster maintained through a joint effort between the University of Tennessee and Oak Ridge National

Laboratory [1]. All source code for the discrete event neural network and neural evolution algorithm can be found in Appendix A.

While this simulator is motivated by biological neural systems, parameter values (e.g. transmission delay, threshold values, refractory period) do not necessarily agree with biological values. The purpose of the simulation is regulation and control of continuous state dynamic systems, not replicating experimental biological data.

3.1 Discrete Event Neural Simulation

The discrete event simulator consists of a graph structure where nodes represent neurons and edges represent axons, dendrites and synapses. Each neuron in the network implements an accumulate-and-fire mechanism. A neuron accumulates pulse events from other neurons. Certain neurons are designated as input neurons and others as output neurons. Input neurons receive inputs external to the network simulation that increase the input neuron accumulator. The pulses generated by output neurons are visible outside the network simulation and used as inputs to continuous time dynamic systems. Neurons can function as both input and output neurons. Neurons that are not input or output neurons are referred to as hidden neurons.

Each neuron has a positive threshold. If the accumulator value of the neuron exceeds its threshold, it will fire a positive pulse. If the accumulator is decreased to a smaller value than the negative value of the threshold, the neuron will fire a negative pulse. Firing a pulse reduces the magnitude of the accumulator by a fixed amount (typically equal to the threshold). If the magnitude of the accumulator value still exceeds the threshold, the neuron will fire until the accumulator value is below the threshold. The rate at which the neuron can fire is limited by the refractory period of the neuron, where the refractory period is the minimum possible time between the firing events of a single neuron.

A neuron has three spatial coordinates which establish distances between neurons. The transmission delay for connected neurons is a linear function of this distance. Connections between neurons can be either inhibitory or excitatory, where inhibitory connections invert the sign of the pulse event. The magnitude of the pulse is determined by a weight term defined for each connection in the network. A neuron can be connected to itself; in this case the transmission delay is set by the refractory period of the neuron.

3.2 Single Neuron Behavior

A single discrete event input neuron, as implemented in this simulation functions as a pulse frequency modulator (PFM), when presented with a continuous-valued continuous time input signal. The behavior of a single neuron is demonstrated below for two different input signals, resulting in a pulse frequency modulated signal. The instantaneous firing rate of the output neuron is low pass filtered using an exponentially weighted moving average (EWMA) to produce a demodulated signal. In both cases, the demodulated signal is an approximation of the original signal.

Figure 3.1 show the reaction of a single neuron that functions as both input and output neuron to a staircase function with steps of increasing magnitude. As the magnitude of the input signal increases, the firing rate of the neuron increases. The input signal is allowed to change, and the system output is reported by the numerical integration routine at multiples of $T_s = 0.01$. The neuron has a threshold $t_h = 100$ and an output gain $g = 1.0$. An EWMA filter with parameter $\alpha = 0.1$ is used to demodulate the signal. With these parameters, the demodulated output of the neuron closely tracks the original signal. Notice that the demodulated signal is not updated until the neuron fires, resulting in a one sample delay, typical of PFM signals.

The behavior of a single neuron with a sampled sinusoidal input ($T_s = 0.01$) is shown in Figure 3.2. This neuron has a threshold $t_h = 10$ and is demodulated by an EWMA low pass filter with parameter $\alpha = 0.9$. The demodulated signal tracks the

modulated signal well, except when the magnitude of the input signal is too small to excite the neuron. This behavior occurs when the input signal transitions from positive to negative. At this point, the accumulator of the neuron is decreased, and the neuron will not fire until its accumulator is decreased sufficiently so that the magnitude of the accumulator exceeds the threshold.

The single neuron behaves as expected for a pulse frequency modulator. When the output parameters, α and g , are tuned appropriately, the demodulated firing events represent the signal being modulated. Hence, the discrete event neuron behaves in an analogous manner to biological neurons. The construction of networks of discrete event neurons is addressed by the neural evolution algorithm.

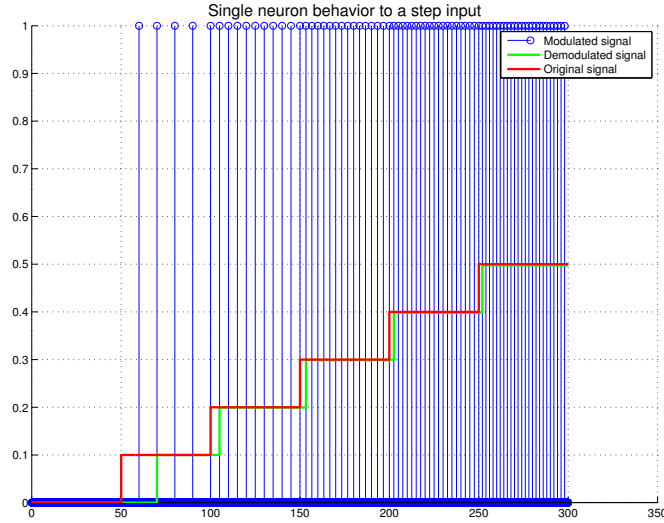


Figure 3.1: Single neuron with staircase input. The demodulated signal tracks the input signal, delayed by one event.

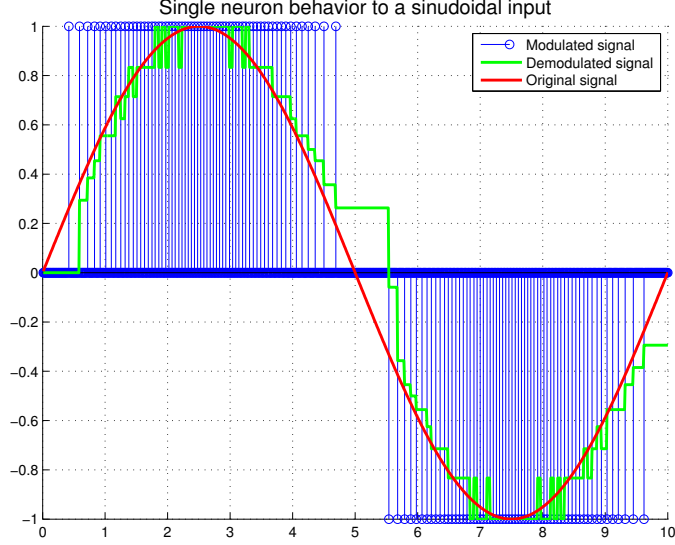


Figure 3.2: Single neuron with sinusoidal input. Insufficient stimulation to the discrete event neuron results in distortions in the demodulated signal.

3.3 Neural Evolution Algorithm

The neural evolution algorithm is used to construct networks of discrete event neurons for use by the discrete event neural simulator. Deterministic or direct methods of network construction are probably not appropriate. Hence a stochastic optimization algorithm is utilized to explore the space of possible network structures. Exploration of this space takes place by applying random mutations to the network.

Mutations to the network can be described as parameter changes or structural changes. The following parameters can be adjusted

- The firing threshold of each individual neuron.
- The minimum time between firing events for each neuron (refractory period).
- The physical location of each neuron on a three dimensional grid with fixed boundaries. The location of neurons determines the communication delay

between the firing event in the parent neuron and the receipt of that event by the child neuron.

- The weight of a connection between two neurons, which scales the magnitude of the pulse.
- The delay per unit distance T_{sc} . This linearly scales the communication delays between all neurons in the network.
- For each of N output neurons, constants $\alpha_1, \dots, \alpha_N$. These constants are used in EWMA low pass filtering the firing rates of the output neurons.
- Constants $g_1, \dots, g_N$. These constants are multiplicative gains applied to the demodulated output of each output neuron.

Mutation of a parameter randomly selects a parameter from the list with equally weighted probabilities. When the value of a parameter is mutated, that value is randomly increased or decreased by a random, uniformly distributed percentage between 0 and 100%.

The following structural modifications can be made to the network.

- Add a neuron. One incoming and one outgoing connection are added to existing randomly selected neurons in the network. The parameters of the neuron are randomly selected.
- Remove a neuron. Incoming and outgoing connections are also removed.
- Add a connection between two randomly chosen neurons that do not have an existing connection. The weight of the connection is initialized to one.
- Remove a connection between two neurons. A neuron with outgoing connections is randomly chosen and a connection from the list of outgoing connections is selected at random for removal.

The neural evolution algorithm used to develop network structures is divided into two main subroutines referred to as the master process and the slave process. The slave process is executed multiple times for each iteration of the master process, but each slave process is independent from other slave processes. This arrangement permits efficient implementation using distributed memory message passing interface (MPI) parallelization by executing slave subroutines on separate processing cores. Source code for the master and slave processes can be found in Appendix A.

The master process implements a genetic algorithm without crossover. A list of variables (Table 3.1) for the pseudo-code (Algorithm 1) for the master process is given. The master process maintains a population $\mathcal{P}$ of n networks. Evaluation of the performance of the network is achieved through a function that independently determines the fitness of each network. The structure of the fitness function is task dependent. In this work, a numerical simulation of the hybrid system consisting of the discrete event simulator interfaced with a continuous time dynamic system is executed. For the systems analyzed in Chapter 4 and Chapter 5, the fitness of the network is determined by a weighted sum of error signals. It is desirable to reduce this error until it reaches a tolerance ϵ . Due to the stochastic aspects of the algorithm, there is no guarantee that the algorithm will reach a specified error tolerance, so in practice it is necessary to provide a limit on the number of iterations the algorithm can perform.

Each iteration of the master process performs a step in a genetic algorithm operating on each population member. A mutation operator modifies the structure of the network and sends that network to a slave process for further refinement. When the slave process has finished, it returns a modified network and fitness for that network, which is added to the existing population. When all of the population members have been processed by a slave process, the new population consists of $2n$ members. These are sorted in order of decreasing performance and the worst n performers are removed, reducing the population to n members. This repeats until the best performing network satisfies a predetermined fitness tolerance.

```

Initialize  $\mathcal{P}$  to  $n$  members
for  $i = 1, \dots, n$  do
     $f_i \leftarrow \text{fitness}(p_i)$ 
end for
 $m \leftarrow 0$ 
while  $\max(F) > \epsilon$  and  $m \leq N$  do
    for all  $p \in \mathcal{P}$  do
         $m \leftarrow m + 1$ 
         $p_m \leftarrow \text{mutate}(p)$ 
         $f_m \leftarrow \text{slave}(p_m)$ 
        Append  $p_m$  to  $\mathcal{P}$ 
        Append  $f_m$  to  $\mathcal{F}$ 
    end for
    Drop  $n$  worst performers from  $\mathcal{P}$ 
end while
return  $p_k \in \mathcal{P} \ni f_k < f \forall f \in \mathcal{F}$ 

```

Algorithm 1: Master process, genetic algorithm.

Table 3.1: A description of variables used in the master process.

Variable	Description
n	The initial size of the population.
$\mathcal{P} = \{p_1, \dots, p_n\}$	The population of best performing networks.
$\mathcal{F} = \{f_1, \dots, f_n\}$	The fitness of each member of $\mathcal{P}$.
ϵ	A tolerance that determines sufficient fitness.
N	Maximum number of iterations to perform.

The slave described by the pseudo-code in Algorithm 2, performs simulated annealing by mutating over network parameters. A description of the variables for this algorithm are in Table 3.2. Each iteration of the slave process mutates

a parameter of the network and evaluates the fitness of the mutated network. In a similar manner to the master process, the slave process must perform a hybrid network/system simulation in order to determine the performance of a proposed network. The algorithm keeps track of the best performing network, but permits continuation of the algorithm with a inferior network with a geometrically decreasing probability T . The annealing temperature T decreases geometrically with rate constant K . If a pseudo-random number falls below the annealing temperature, the algorithm continues to evaluate that network, even if it performs worse than the networks in previous iterations. Since the annealing temperature decreases with each iteration, the probability of this occurring decreases as the algorithm continues. After N iterations, the slave process returns the best performing network to the master process.

Receive network p_m from master.

$b \leftarrow \text{fitness}(p_m)$

$T \leftarrow 1.0$

$y \leftarrow p_m$

for $i \leftarrow 0$ to N **do**

$T \leftarrow K * T$

$z \leftarrow \text{mutate}(y)$

$f \leftarrow \text{fitness}(z)$

if $f < b$ **then**

$p_m \leftarrow z$

$y \leftarrow z$

$b \leftarrow f$

else if $\text{rand}() > T$ **then**

$y \leftarrow p_m$

end if

end for

return b, p_m

Algorithm 2: Slave process, simulated annealing.

Table 3.2: A description of variables used in the slave processes.

Variable	Description
p_m	The population member sent from the master process.
T	The annealing temperature.
$K \in (0, 1)$	An annealing constant.
N	Number of iteration of simulated annealing to perform.

3.4 Network Visualization

A custom visualization tool is utilized to view network structures. An example is provided in Figure 3.3. Blue nodes indicate input neurons, red nodes are output neurons, and green nodes are hidden neurons. Blue connections are excitatory and red connections are inhibitory pathways. The thickness of the connection indicates the strength of the connection between two neurons. In the example the input neuron is connected to a hidden neuron by an inhibitory connection. This hidden neuron is connected to the output neuron by an excitatory connection. With appropriately chosen thresholds and weights, an input applied to the network will generate an output pulse. The third connection, represented by a thicker line represents an excitatory connection with a greater weight. When the output neuron fires, it stimulates the input neuron, and the process starts again. If the parameters are chosen appropriately, this establishes a sustained firing pattern similar to a central pattern generator.

A detailed description of the parameters of neurons in this network can be found in Table 3.3. For each neuron, the table list values for the spatial coordinates (x, y, z) , the threshold T_h , and the refractory period T_{rf} . For output neurons, parameter values are provided for the output gain g and the output filter coefficient α . The time scale T_{sc} is a property of the network and must also be specified to fully define the network. Table 3.4 lists the connections present in the network, indicated by a starting neuron (parent) and a terminal neuron (child), as well as the weight of that connection.

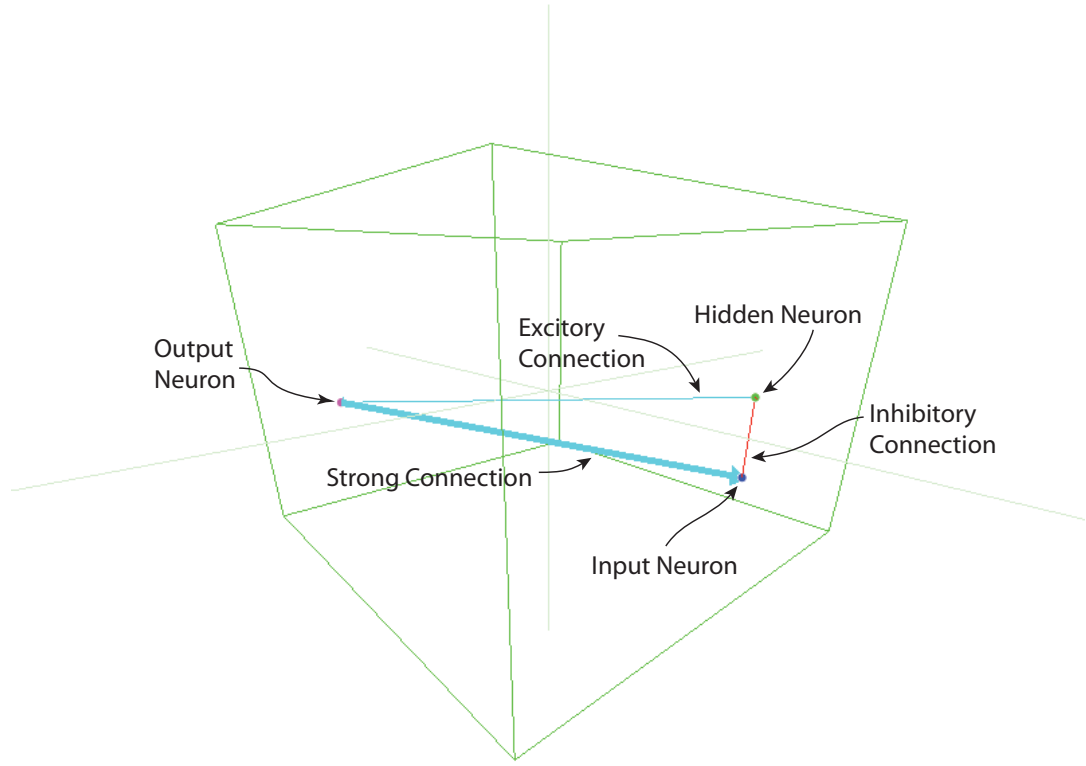


Figure 3.3: A sample network observed through the network visualization utility.

Table 3.3: Parameter details for the network visualization example.

Neuron	Type	x	y	z	T_h	T_{rf}	g	α
1	input	0.443	-0.081	-0.11	1.0	0.001	—	—
2	hidden	-0.3	-0.034	0.09	5.458	0.001	—	—
3	output	-0.3	-4.32	0.0	9.34	0.001	1.0	0.866

Table 3.4: Connection details for the network described in Table 3.3. This network forms a complete cycle.

Parent	Child	Weight
1	2	1.0
2	3	1.0
3	1	5.0

3.5 Summary

This chapter presents a model neural simulation that emphasizes the pulses used in communication between neurons. These pulses are treated as discrete events. With a discrete event neural simulator, a method is provided for building networks of discrete event neurons, as well as interfacing discrete event neural simulation to simulations to numerical simulations of systems with continuous time dynamics. The remainder of this work explores the interactions of the discrete event neural simulation with continuous time dynamics.

Chapter 4

Harmonic Oscillator

The chapter explores the interaction between the discrete event network simulation and a model of a harmonic oscillator. The network provides closed loop feedback to the system (Figure 4.1). Its purpose is to regulate the states of the harmonic oscillator to zero. Networks are initialized according to the structure of the harmonic oscillator: two input neurons are provided for each state of the harmonic oscillator and one output neuron is provided for the control signal to the system.

For the harmonic oscillator described by the differential equation in Equation 2.16 with output $y(t)$, let $x_1(t) = y(t)$, $x_2(t) = \dot{y}(t)$, and $\underline{x} = (x_1, x_2)^T$. Then this system has a state space model

$$\dot{\underline{x}}(t) = A\underline{x}(t) + Bu(t) \quad (4.1)$$

where

$$A = \begin{pmatrix} 0 & 1 \\ -\frac{k}{m} & 0 \end{pmatrix}$$

and

$$B = \begin{pmatrix} 0 \\ \frac{1}{m} \end{pmatrix}$$

This is a linear time invariant system. In this chapter the constants of the harmonic oscillator are $k = 1$ and $m = 0.1$, yielding a frequency of oscillation

$\frac{1}{2\pi}\sqrt{\frac{k}{m}} = 0.503s^{-1}$. Full state feedback is implemented by providing an input neuron for each of the two states x_1 and x_2 ; one output neuron, when demodulated, provides the input to the system u . The control goal of the feedback network is to stabilize and regulate the states of the harmonic oscillator around the equilibrium point $\underline{x} = \underline{0}$.

The signal generated by the output neuron is demodulated in the same manner as a PFM signal using the methods discussed in Section 2.2.1. All additional neurons, synapses, and relevant network parameters are evolved using the network evolution algorithm from Section 3.3 using two computational nodes, each with 2.8 GHz dual quad core AMD Opteron processors with 8GB RAM (8 cores/node and 16 cores total).

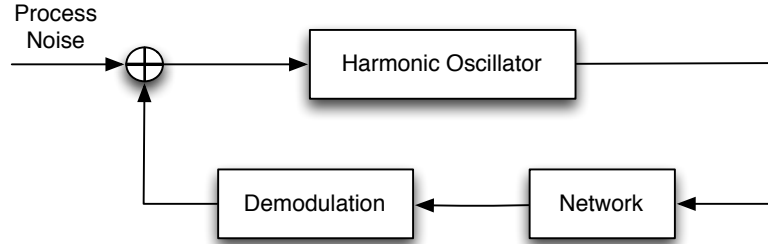


Figure 4.1: Block diagram of the harmonic oscillator with discrete event network feedback.

The behavior of the unforced harmonic oscillator is compared to the closed loop system without noise and in the presence of process noise. Band-limited noise is generated by low pass filtering zero mean and unit variance Gaussian white noise using an exponentially weighted moving average (EWMA). In the absence of an explicit gain term, the EWMA smoothing constant $\alpha \in (0, 1]$ controls the severity of the noise process. A small value of alpha (say $\alpha = 0.01$) produces a smoother random signal than a larger value (say $\alpha = 0.9$). An $\alpha = 1.0$ corresponds to an unfiltered noise process. A small value of α corresponds to a low pass filter with a small cutoff frequency resulting in a smoother noise process with less variance. Sample noise process are provided for reference in Figure 4.2.

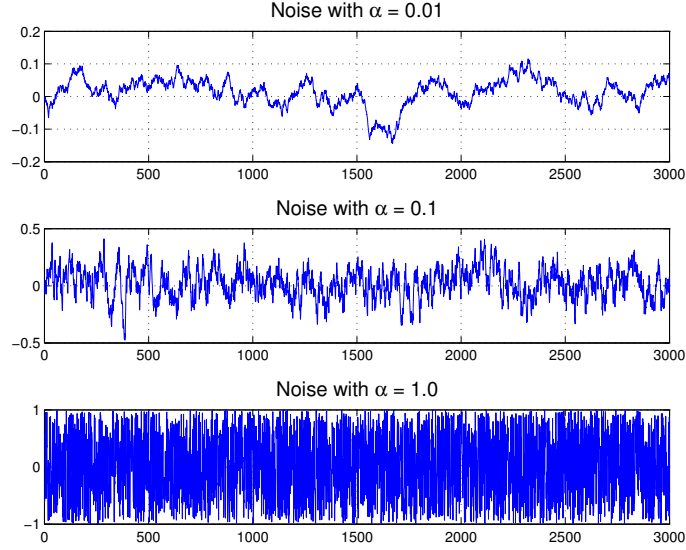


Figure 4.2: Examples of band-limited noise processes with differ filter parameters.

Performance is measured by the mean-squared-error (MSE) of x_1 over time. As defined below, this measures the average deviation of this state from zero. The MSE of N samples $x_1(1), x_1(2), \dots, x_1(N)$ is given by [44]

$$MSE(x_1) = \frac{1}{N} \sum_{n=1}^N x_1^2(n) \quad (4.2)$$

Without noise this is a simple problem, and simple structures are found by the evolutionary algorithm. Process noise, however, requires more complex network structures to effectively maintain minimal system response to the noise.

4.1 Unforced System

The trajectory of of the unforced harmonic oscillator is shown in Figure 4.3, with the initial condition $x_1(0) = 1$. The states x_1 and x_2 oscillate sinusoidally. The objective of the control problem is to drive the states $\underline{x}$ to $\underline{0}$ by minimizing the MSE criterion. For the non zero initial conditions used in this study, an MSE of zero is not feasible.

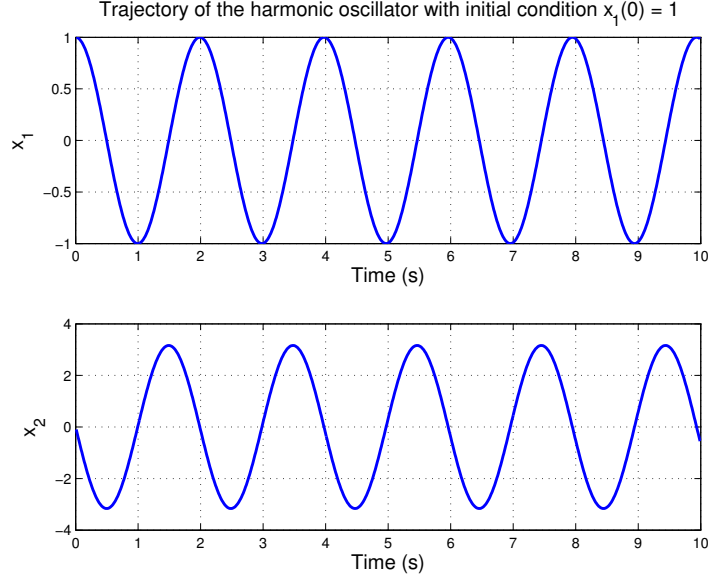


Figure 4.3: Behavior of the unforced harmonic oscillator. This system oscillates sinusoidally.

4.2 Network Optimization Examples

The neural evolution algorithm generates networks capable of driving the harmonic oscillator to small states. In these experiments, the neural evolution algorithm was run until a network satisfying a tolerance $MSE(x_1) = 0.009$ was found. Simple network structures emerge when this optimization takes place without noise. The simple structures do not perform well in the presence of noise. When the optimization takes place in the presence of small amounts of process noise, complex network structures develop with more neurons, synapses, and cyclic connections. These networks exhibit improved performance with and without noise. The examples presented in this section are typical representations of the networks produced under the given environments. A statistical analysis of the results of repeated trials of the neural evolution algorithm is presented in Section 4.3.

Network 1 (Figure 4.4, parameter details provided in Table 4.1) provides an example of a network optimized in the absence of any process noise. As can be observed, this results in a simple structure with no cyclic connections. In fact, this network disregards feedback from one of the states of the system (in this case x_2).

Simulation of the closed loop system with network 1 is shown in Figure 4.5. This network has a time scale $T_{sc} = 0.20861$. This network is capable of generating an input signal to the system that drives the error to a small value ($MSE = 0.0088078$). Note that the states are never driven identically to zero. This is an effect of the PFM characteristics of the input neurons. The feedback signal from the system to the network decreases to the point where it no longer excites the network. The neural evolution algorithm required only two generations to discover this network. This simple network does not, however, handle even small amounts of process noise. Process noise with $\alpha = 0.01$ (Figure 4.6) is enough to consistently destabilize the closed loop system.

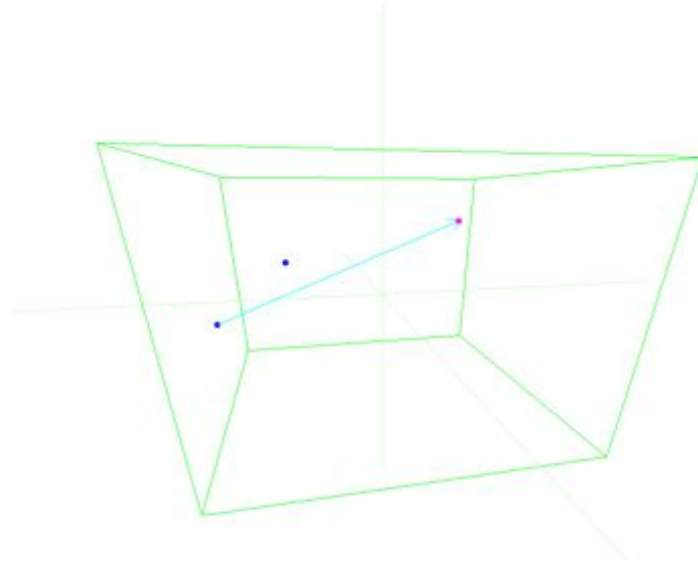


Figure 4.4: Network construction in the absence of process noise results in simple network structures.

Table 4.1: Details of the feedback network for the harmonic oscillator optimized without noise.

Neuron	Type	x	y	z	T_h	T_{rf}	g	α
1	input	0.5	0.1	0.0	1.398	0.0001	—	—
2	input	0.272	-0.231	-0.366	5.43	0.0001	—	—
3	output	-0.047	-0.346	-0.463	9.71	0.0001	2.50253	0.688474

Table 4.2: Connection details for the network described in Table 4.1. Only one connection is formed.

Parent	Child	Weight
1	3	1.34771

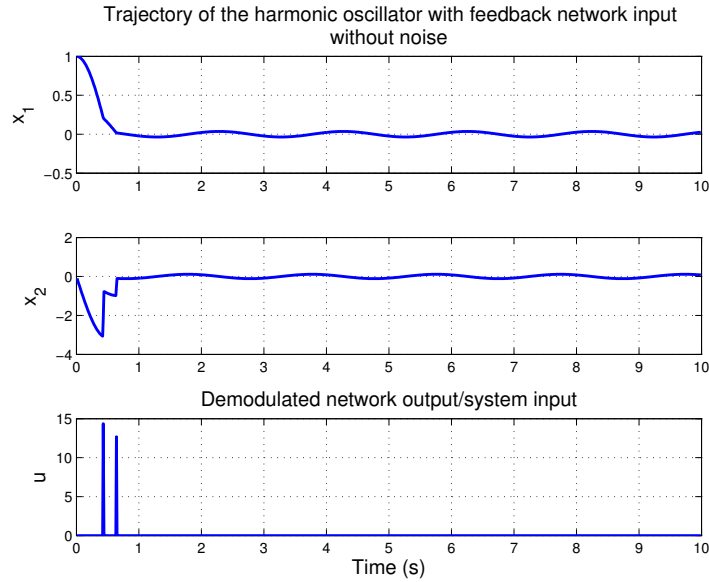


Figure 4.5: Behavior of the closed loop harmonic oscillator without noise using a feedback network developed in the absence of process noise. This approach generates a simple control signal to the system.

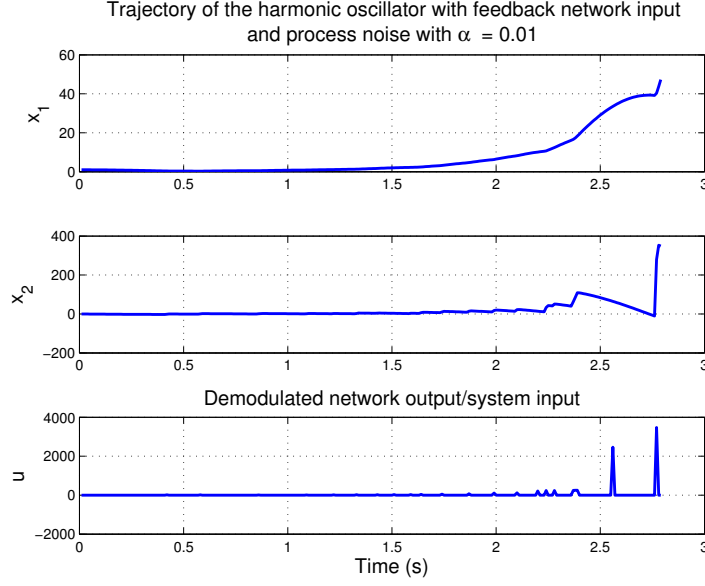


Figure 4.6: Behavior of the closed loop harmonic oscillator with a small amount of process noise using a feedback network developed in the absence of process noise. The closed loop system is unstable with even a small amount of noise.

Network 2 (Figure 4.7) was developed using the neural evolution algorithm in the presence of a small amount of process noise ($\alpha = 0.01$) and has a time scale $T_{sc} = 0.000603$. The algorithm found this network in 13 generations with $MSE = 0.00825988$. This network performs well without process noise (Figure 4.8). Without noise, a distinct quasi-periodic output is generated by the network, even when the magnitude of the states decreases to a small value. The network also performs well for a small amount of process noise with $\alpha = 0.01$ (Figure 4.9) and for a larger amount of process noise with $\alpha = 1.0$ (Figure 4.10). A sustained activity is visible in the presence of process noise. The structure of this activity is less apparent for the increased noise, since the noise is large enough to generate much greater activity in the network. Average error over 50 simulations is shown in Table 4.5. Network 2 successfully controls the harmonic oscillator in the presence of non trivial noise processes, without exhibiting unstable behavior.

This network is considerably more complex than the previous network and presents a greater analysis challenge. The state x_1 excites input neuron 1, which forms a cyclic connection with hidden neuron 4. This structure is primarily responsible for generating the sustained CPG-like behavior observed in the output of the closed loop system. There is an excitatory path from input neuron 1 to input neuron 2 and to output neuron 3 through hidden neuron 5. Hidden neuron 5 is inhibited, however, by hidden neuron 6. Input neuron 2, which receives input from state x_2 , inhibits output neuron 3. Output neuron 3 is also inhibited by hidden neuron 4 through hidden neuron 7. There are no cyclic paths stimulating input neuron 2. Once x_2 achieves a small value, its influence on the network becomes minimal.

The cyclic connection between input neuron 1 and hidden neuron 4 is the only cyclic pathway created by the neural evolution algorithm in this network. All other connections are feedforward connections that merely introduce delay to the control signal. An interesting result can be observed from the demodulated network output (i.e. the control signal to the harmonic oscillator). The sustained quasi-periodic behavior of the network output occurs at approximately the same frequency as the state x_2 , where the sign of the network signal is negative that of the state. The network maintains a model of the derivative of the state x_1 even when the state x_2 is driven to a small value. The network has appeared to learn a model of the harmonic oscillator using a cyclic connection (between neuron 1 and neuron 4) and delay.

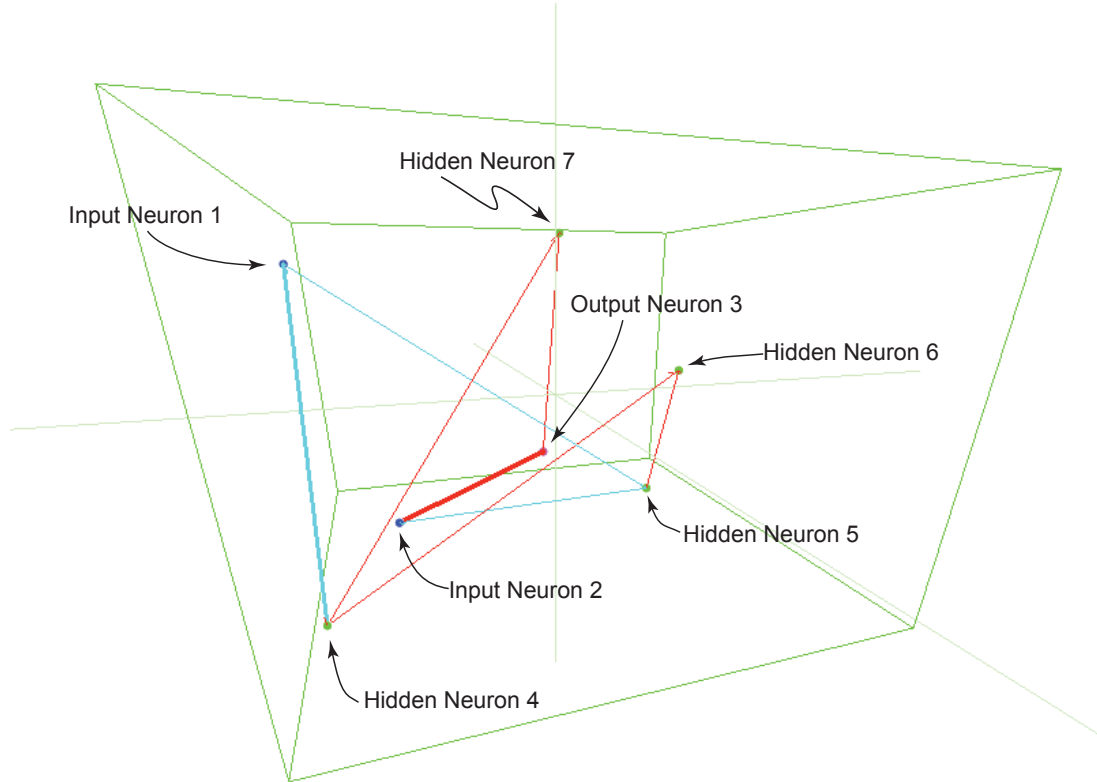


Figure 4.7: Network developed in the presence of process noise.

Table 4.3: Details of the feedback network for the harmonic oscillator optimized in the presence of a small amount of process noise.

Neuron	Type	x	y	z	T_h	T_{rf}	g	α
1	input	-0.0554	-0.353	0.474	4.56	0.0001	—	—
2	input	0.299	0.161	0.336	1.38	0.0001	—	—
3	output	-0.122	0.233	-0.00961	3.813	0.0001	0.180	2.61e-5
4	hidden	0.444	0.246	0.433	0.261	0.390	—	—
5	hidden	0.296	0.153	-0.0668	0.284	0.797	—	—
6	hidden	0.0485	-0.639	-0.256	0.0958	0.371	—	—
7	hidden	-0.135	-0.448	-0.0525	0.368	0.692	—	—

Table 4.4: Connection details for the network described in Table 4.3. A cyclic connection is present between neuron 1 and neuron 4.

Parent	Child	Weight
1	4	-0.00304
1	5	0.200
2	3	-4.28
4	6	-0.252
4	7	-0.309
4	1	2.55
5	2	0.0675
6	5	-0.0298
7	3	-0.296

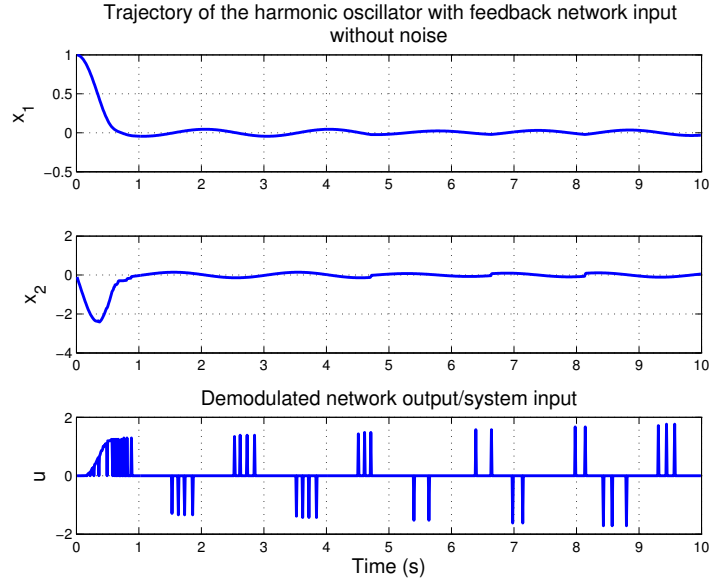


Figure 4.8: Harmonic oscillator with feedback using a network developed with a small amount of process noise. Without noise, the network sustains CPG-like behavior when the input signal is negligible.

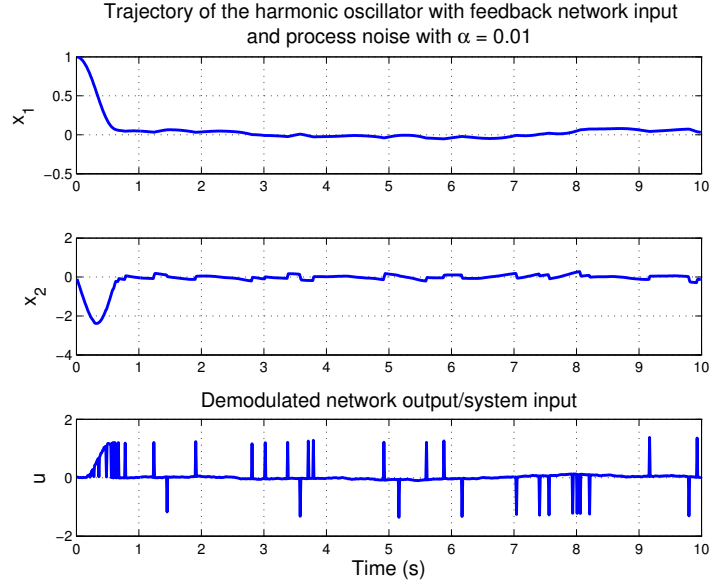


Figure 4.9: Performance of the harmonic oscillator with process noise with $\alpha = 0.01$ using a feedback network developed in the presence of process noise. Sustained behavior is still observed despite the presence of process noise.

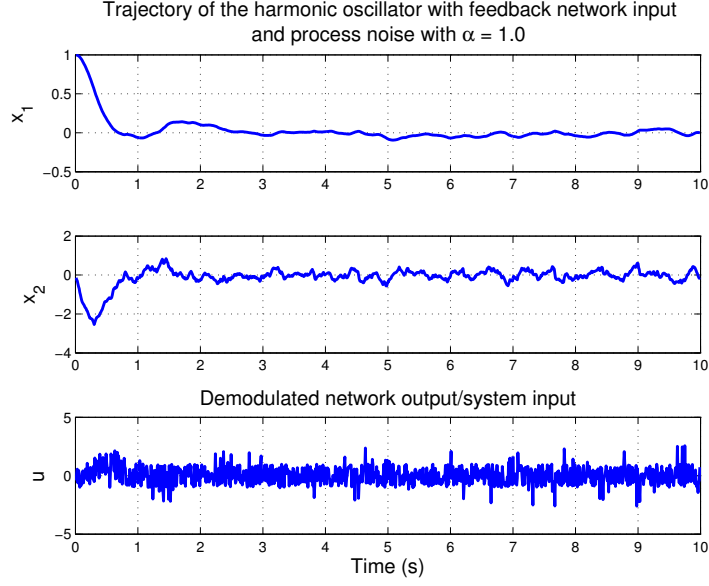


Figure 4.10: Performance of the harmonic oscillator with process noise with $\alpha = 1.0$ using a feedback network developed in the presence of process noise. The feedback network stabilizes the system in the presence of unfiltered process noise.

α	Average MSE
0	0.00891719
0.01	0.00949
1.0	0.0124

Table 4.5: Average performance of Network 2 over 50 simulations shows favorable performance even in the presence of process noise.

4.3 Neural Evolution Algorithm Statistics

This section examines statistics regarding the efficiency of the neural evolution algorithm. Statistics are averaged over 50 optimization runs for each case of population size from one to fifteen. As can be seen in Figure 4.11, having a sufficiently large population size results in fewer generations needed to explore the space of

possible network structures. When multithreading is available to allow separate processes to explore the parameter space for a population of fixed network structures, the average execution time and average are required to find a solution decreases with the average number of generations. Increasing the population size beyond eight members does not significantly improve the performance of the algorithm for this application. Having more than one or two population members is critical to finding a solution efficiently.

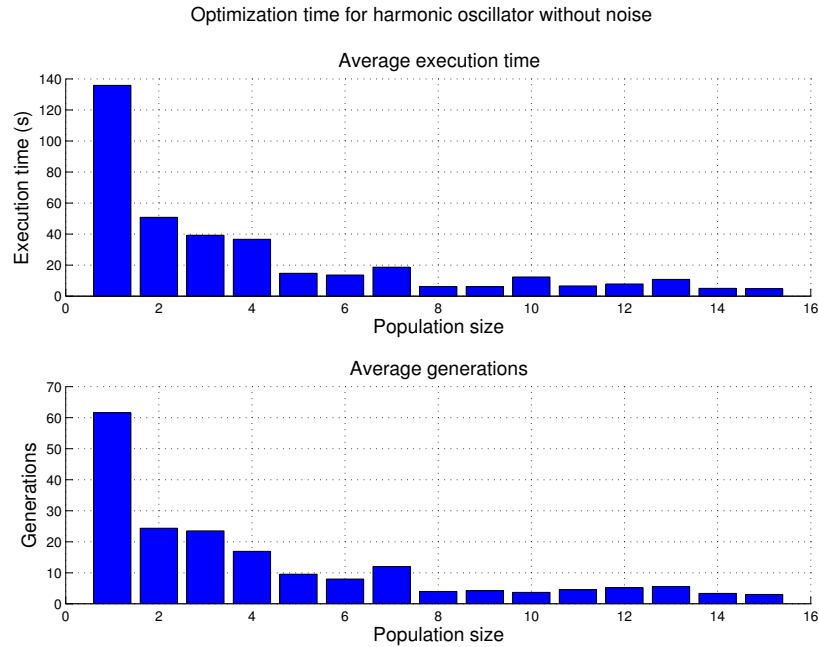


Figure 4.11: Average execution time and average generations required to find a network solution decrease as the population size increases.

In terms of computational efficiency, it is desirable to find a network with a minimal number of neurons and synapses. The average numbers of networks evaluated with a given number of neurons (Figure 4.12) and a given number of synapses (Figure 4.13) are observed. Increasing the population size results in a more complete search of networks with fewer neurons and synapses. This implies that an insufficient

population size will result in structural mutations without exploring enough of the parameter space of each structure.

Without noise, the neural evolution algorithm tends to explore network structures with fewer than 10 neurons when the population size increases beyond 8 slave processes. Similar behavior is expressed in the average number of synapses explored. Increasing the number of slave process beyond 8 reduces the number of synapses explored to less than 10 in most cases.

Average number of neurons explored for harmonic oscillator without noise

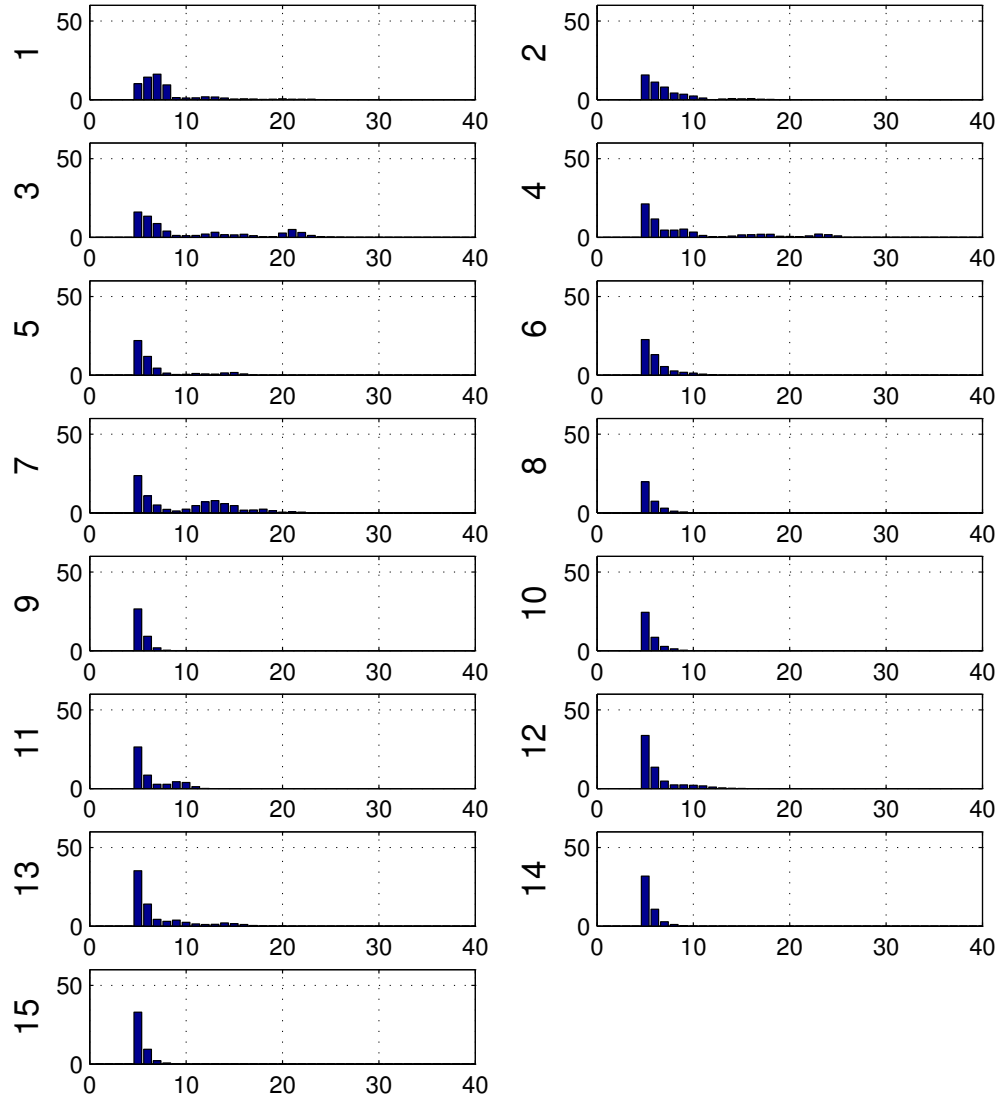


Figure 4.12: The average number of neurons explored by the neural evolution algorithm is reduced on average when the population size is increased, resulting in less complex network structures.

Average number of synapses explored for harmonic oscillator without noise

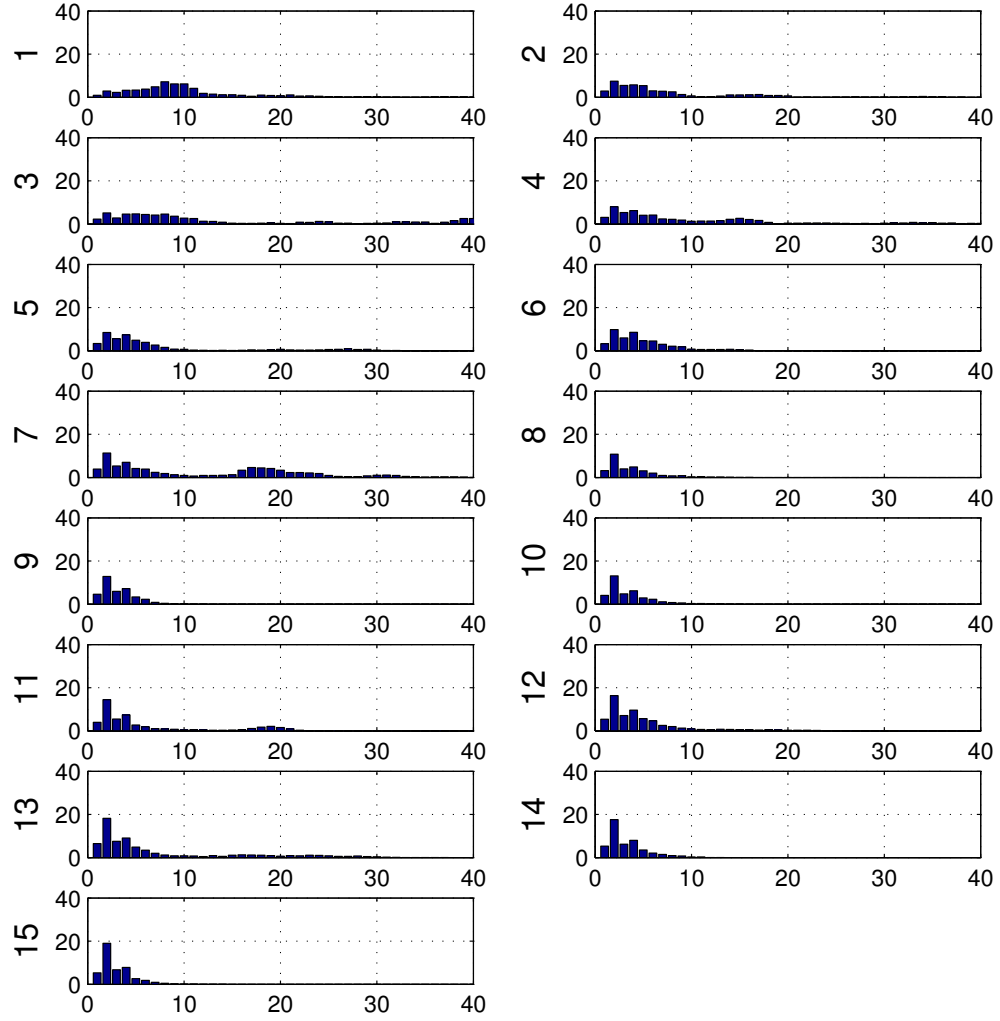


Figure 4.13: The average number of synapses explored by the neural evolution algorithm is reduced on average when the population size is increased, resulting in less complex network structures.

4.4 Summary

The neural evolution algorithm can generate network structures that the discrete event neural simulator can utilize as feedback controllers for the harmonic oscillator. Simple structures emerge when the algorithm develops networks in the absence of noise; more complex networks develop when the evolution process takes place in the presence of process noise. Use of the simulator with process noise significantly improves the performance of the closed loop system with and without process noise. A sample network demonstrates the capability of a more complex network structure to develop a model to use for control of the harmonic oscillator.

Chapter 5

Control of the Cart-and-Pendulum

The cart-and-pendulum (Figure 5.1) is a common formulation of the inverted pendulum control problem. A pendulum is fixed at a pivot to a cart that can move horizontally. Stabilization of the pendulum is achieved by applying a force to the cart, moving the cart such that it remains under the pendulum. A detailed model for the cart-and-pendulum is provided. A closed loop feedback control strategy that combines nonlinear energy control and linear quadratic regulation is provided. Introduction of the neural simulator operating on the error signal of the closed loop system improves performance significantly without process noise and in the presence of process noise.

5.1 Model

The canonical cart-and-pendulum (Figure 5.1) is used in this work. The chosen numerical parameter values are shown in Table 5.1. The differential equations describing this system are

$$I\ddot{\theta} = mgL \sin \theta - mL^2\ddot{\theta} - mL\ddot{y} \cos \theta \quad (5.1)$$

$$M\ddot{y} = F - m(\ddot{y} + L\ddot{\theta} \cos \theta - L\dot{\theta}^2 \sin \theta) - k\dot{y} \quad (5.2)$$

where θ is the angular displacement of the pendulum from the vertical axis, y is the linear displacement of the cart, M is the mass of the cart, m is the mass of the pendulum, L is the distance from center of gravity of the pendulum to the pivot, I is the moment of inertia of the pendulum, and k is a friction coefficient [34]. Let $x_1 = \theta$, $x_2 = \dot{\theta}$, $x_3 = y$ and $x_4 = \dot{y}$ be the states of the system, $\underline{x} = (x_1, x_2, x_3, x_4)^T$, and $u = F$ be the input. Then the system has the state space form

$$\dot{\underline{x}} = \underline{f}(\underline{x}, u) = (f_1(\underline{x}, u), f_2(\underline{x}, u), f_3(\underline{x}, u), f_4(\underline{x}, u)) \quad (5.3)$$

where $f_1(\underline{x}, u) = x_2$, $f_3(\underline{x}, u) = x_4$, and

$$\begin{pmatrix} f_2(\underline{x}, u) \\ f_4(\underline{x}, u) \end{pmatrix} = \frac{1}{\Delta} \begin{pmatrix} m + M & -mL \cos x_1 \\ -mL \cos x_1 & I + mL^2 \end{pmatrix} \begin{pmatrix} mgL \sin x_1 \\ u + mLx_2^2 \sin x_1 - kx_4 \end{pmatrix}$$

where $\Delta = (I + mL^2)(m + M) - m^2L^2 \cos^2 x_1$.

This system presents a more challenging control problem than the harmonic oscillator. There are four states to control via a single input. Stabilization involves minimizing the angular displacement of the pendulum, while maintaining a neutral position of the cart.

Performance $P(x_1, x_3)$ of this system is measured using the weighted sum of the mean-squared-error (Equation 4.2) of the angular displacement and the linear displacement such that

$$P = w_1 MSE(x_1) + w_3 MSE(x_3) \quad (5.4)$$

where w_1 and w_3 are the weights for the respective states. For the applications presented here, equal weight is give to the angular position and the linear position by letting $w_1 = 0.5$ and $w_2 = 0.5$.

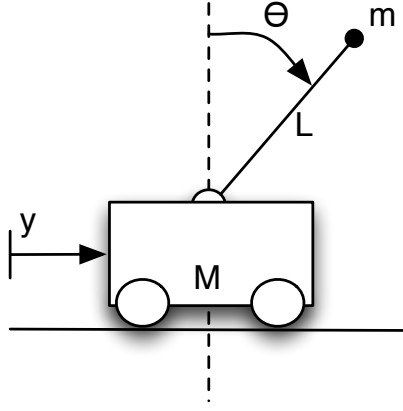


Figure 5.1: The inverted pendulum on a cart.

Table 5.1: Parameters used for the cart-and-pendulum system.

Parameter	Value	Units
M	10	kg
m	2	kg
L	0.5	m
k	0.1	$\frac{kg}{s}$
g	9.81	$\frac{m}{s^2}$
I	$\frac{1}{3}mL^2$	
k_p	0.9	$\frac{s}{m}$
ϵ_1	8.53	—
ϵ_2	0.599	—

5.2 Cart-and-Pendulum Control Strategy

Nonlinear swing-up control is implemented utilizing an energy based method [67, 54]. The total energy E of the pendulum (rotational kinetic energy and potential energy) is given by

$$E = \frac{1}{2}(I + mL^2)\dot{\theta}^2 + mgL(1 + \cos \theta) \quad (5.5)$$

Taking the derivative of this expression and substituting the equation for angular acceleration gives

$$\dot{E} = -mL\ddot{\theta}\dot{\theta}\cos\theta \quad (5.6)$$

Hence the energy of the pendulum can be controlled by applying a force to the cart, consequentially changing the linear acceleration $\ddot{y}$ of the cart.

Control of the cart is achieved using the control law

$$F = k_p(E - E_0)\dot{\theta}\cos\theta \quad (5.7)$$

where $k_p > 0$ is a proportionality constant, and E_0 is the desired energy of the system. In this case, the desired energy $E_0 = 2mgL$ corresponding to the potential energy of the pendulum at the unstable equilibrium. Define the Lyapunov function

$$V = \frac{1}{2}(E - E_0)^2 \quad (5.8)$$

This has the derivative

$$\dot{V} = (E - E_0)\dot{E} = -k_p L \left((E - E_0)\dot{\theta}\cos\theta \right)^2 \quad (5.9)$$

Then V is positive semidefinite and $\dot{V}$ is negative semidefinite, except at certain points in the state space, in particular at $(\theta, \dot{\theta}) = (0, 0)$. At this point, the Lyapunov equations will not be satisfied for the given control law.

Closed loop control can be achieved by switching to a linear quadratic regulator (LQR) when the states of the system reach a small value and an approximate linear model linearized around the unstable equilibrium point is valid. Linearizing the nonlinear system at the unstable equilibrium point $\underline{x} = \underline{0}$ results in a linear state

space system (Equation 2.18) where

$$A = \begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 \\ 16.8171 & 0.0 & 0.0143 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \\ -1.4014 & 0.0 & -0.0095 & 0.0 \end{pmatrix}$$

and

$$B = \begin{pmatrix} 0.0 \\ -0.1429 \\ 0.0 \\ 0.09052 \end{pmatrix}$$

The LQR controller is calculated by MATLAB generating a linear control law $u = -\underline{K}x$. Weighting matrices Q and R are chosen such that the cart and pendulum position errors are penalized:

$$Q = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

and $R = 1$. The nonlinear cart and pendulum system linearized near the unstable equilibrium point produces a feedback term $K = (-258.9975, -63.5652, -1.1050, -5.1523)$. It is important to note at this point, that both the energy control method and the LQR controller require state feedback as implemented. While this is sufficient for the simulation presented here, practical implementation of these controllers would require the use of a state estimator.

The switching criteria implemented is the same used by [67]. For two small empirically determined constants $\epsilon_1, \epsilon_2 > 0$, if both $|E - E_0| < \epsilon_1$ and $1 - \cos \theta < \epsilon_2$ are satisfied, then the control algorithm switches to an LQR controller [5]. The first

condition ensures that the energy of the pendulum is minimal. The desired energy E_0 is not uniquely achieved at the unstable equilibrium point, hence the second condition ensures that the pendulum is near the unstable equilibrium point.

5.3 Improved Performance Using the Neural Simulator

A neural network simulation is utilized to improve the performance of the closed loop system in the presence of process noise (Figure 5.2). Band-limited process noise is generated by low pass filtering Gaussian white noise using a low pass filter implemented by an exponentially weighted moving average with a filter parameter $\alpha = 0.5$. A multiplicative gain is applied to the noise process to increase the magnitude of the noise. The gain values chosen are 1.0, 1.5, 2.0. In the absence of the neural network, increasing the process noise gain decrease the performance of the closed loop system. Including a discrete event neural system in the manner described in Figure 5.2 improves the performance of the system.

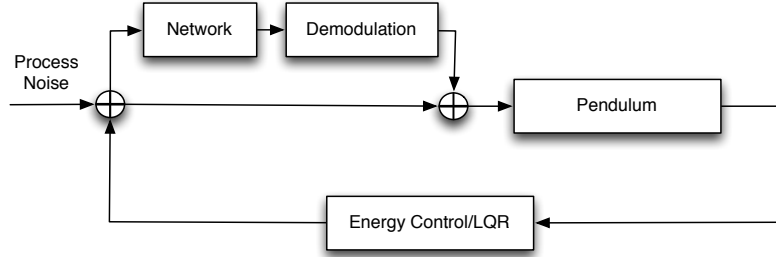


Figure 5.2: Block diagram of the closed loop system including a neural network simulation that improves the performance of swing-up control of the inverted pendulum.

A tiered approach is taken for network construction. The neural network is initialized by applying the neural evolution algorithm without process noise. That

network is then used as a seed for the next optimization task, which applies process noise with a gain of 1.0. This is repeated for process noise gains of 1.5 and 2.0. For process noise with a gain of over 2.0, the closed loop system without the network is unstable. The neural evolution algorithm is unable to stabilize the closed loop system in this case. Since the noise varies for each simulation, the performance achieved by a particular network structure is averaged over ten separate simulations in the neural simulation algorithm. In each case, the constructed network is also able to control the pendulum for tested gains less than the gain that was used to develop the network.

Average error for the different gain terms can be seen in Figure 5.3. The results are averaged over 50 simulations in the presence of processes noise with the given gains. For each process noise gain (0.0 indicating no process noise), there are two bars indicating the average performance of the controller with and without the network component. Error bars specify one standard deviation from the average. System performance is improved for gains of 1.0 and 1.5. A gain of 2.0 represents the limit at which the system can perform without being consistently destabilized by the process noise. A gain greater than 2.0 results in unstable behavior with or without the network.

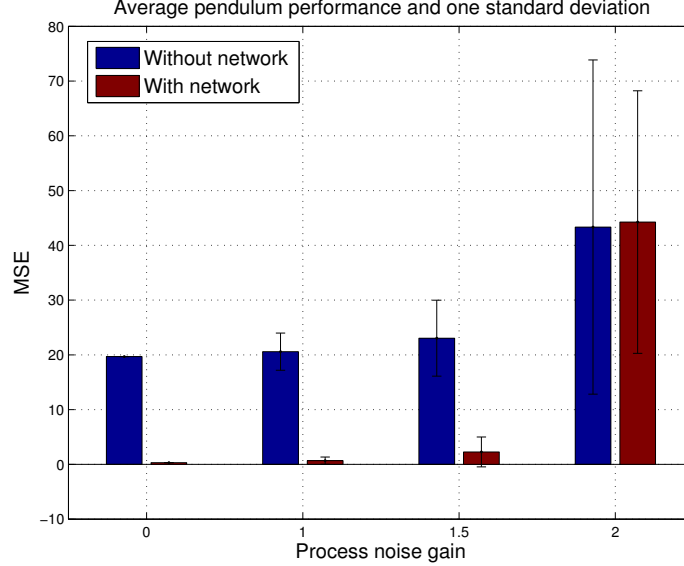


Figure 5.3: An appropriately tuned network improves the robustness of the closed loop system.

Example final structures derived during each tier of the neural evolution algorithm are shown in Figures 5.4–5.7. These are typical networks that produce average errors representative of the average errors shown in Figure 5.3. The first three systems have a common excitatory connection from the input neuron to the output neuron and an inhibitory connection from the output neuron back to the input neuron. Self-connections develop in each of these networks. The weights, however, are relatively small and have little impact on the network behavior. The network developed without process noise has a time scale of $T_{sc} = 1.0$. For the network developed in the presence of process noise with a gain of 1.0, the time scale decreases significantly to $T_{sc} = 0.000394$. For a gain of 1.5 the time scale remains small with $T_{sc} = 0.000656$. Increasing the gain to 2.0 does not significantly influence the time scale, which is $T_{sc} = 0.000764$.

When the process noise gain is at 2.0, the limit of stable behavior, additional structure is needed to achieve this performance. The network structure in Figure 5.7 shows several cyclic connections. These cycles connect neurons 0–2–1–0, 0–1–0, and

1–3–2–1. Neuron 4, and all connection to neuron 4, can be regarded as extraneous structure since neuron 4 has no outgoing connection.

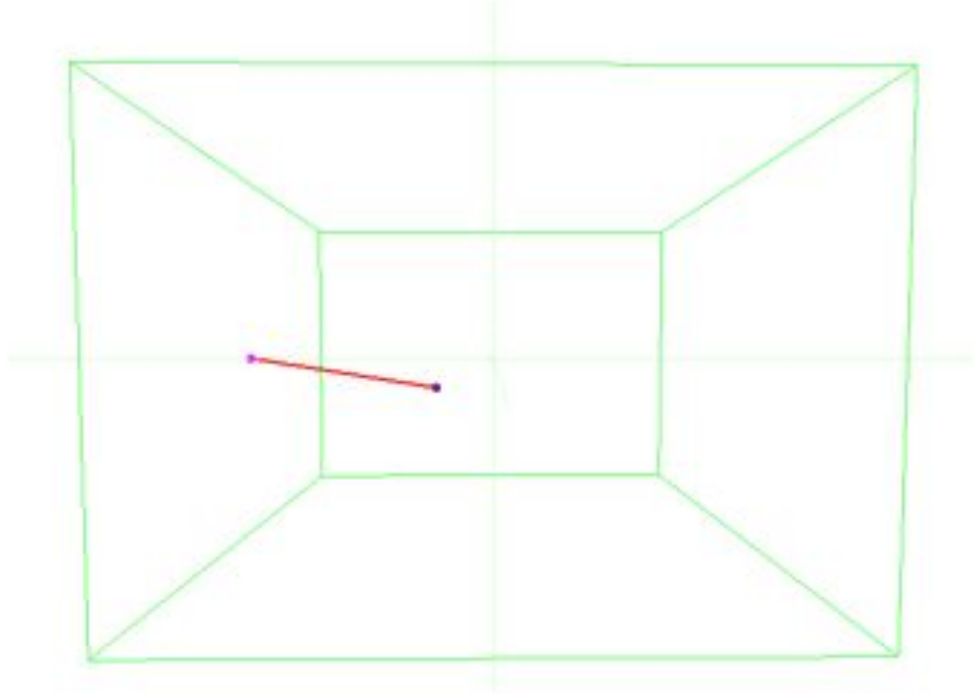


Figure 5.4: A simple network structure developed when the neural simulation algorithm proceeds in the absence of process noise.

Table 5.2: Parameter values for the network in Figure 5.4.								
Neuron	Type	x	y	z	T_h	T_{rf}	g	α
0	input	0.142	0.112	0.270	20	0.0001	–	–
1	output	0.5	0.0	0.0	1.0	0.0001	0.00917	0.10

Table 5.3: Connection values for the network in Figure 5.4.

Parent	Child	Weight
0	0	-0.7111
0	1	1.0
1	0	-1.89

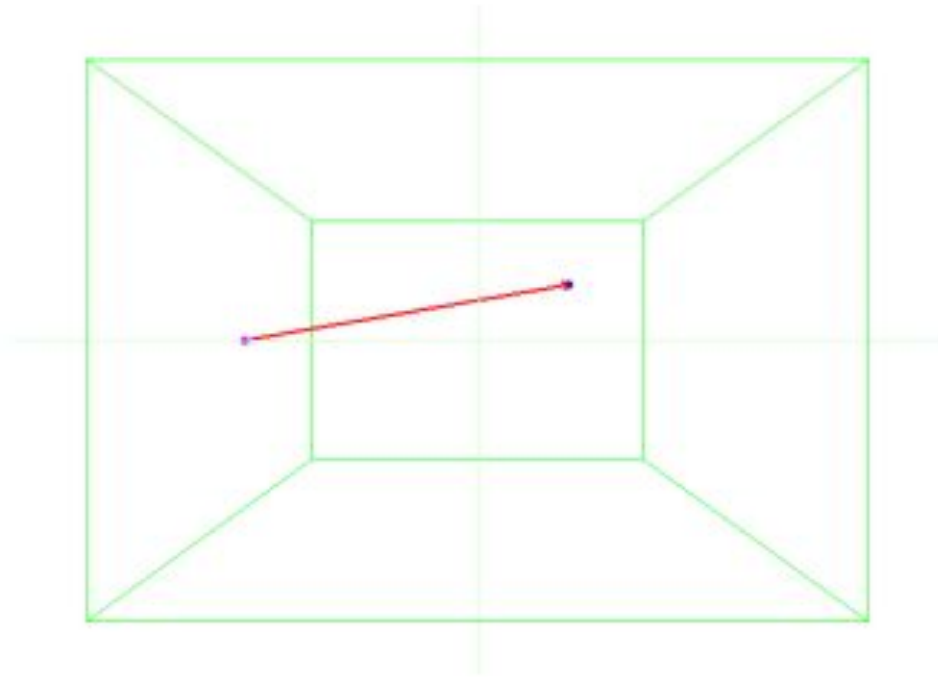


Figure 5.5: Neural network structures for neural evolution for process noise with a gain of 1.0

Table 5.4: Parameter values for the network in Figure 5.5.

Neuron	Type	x	y	z	T_h	T_{rf}	g	α
0	input	-0.197	-0.168	0.009	20	0.0001	—	—
1	output	0.5	0.0	0.0	0.978	0.0001	0.0064	0.608

Table 5.5: Connection values for the network in Figure 5.5.

Parent	Child	Weight
0	0	-0.711
0	1	0.567
1	1	0.0127
1	0	-1.898

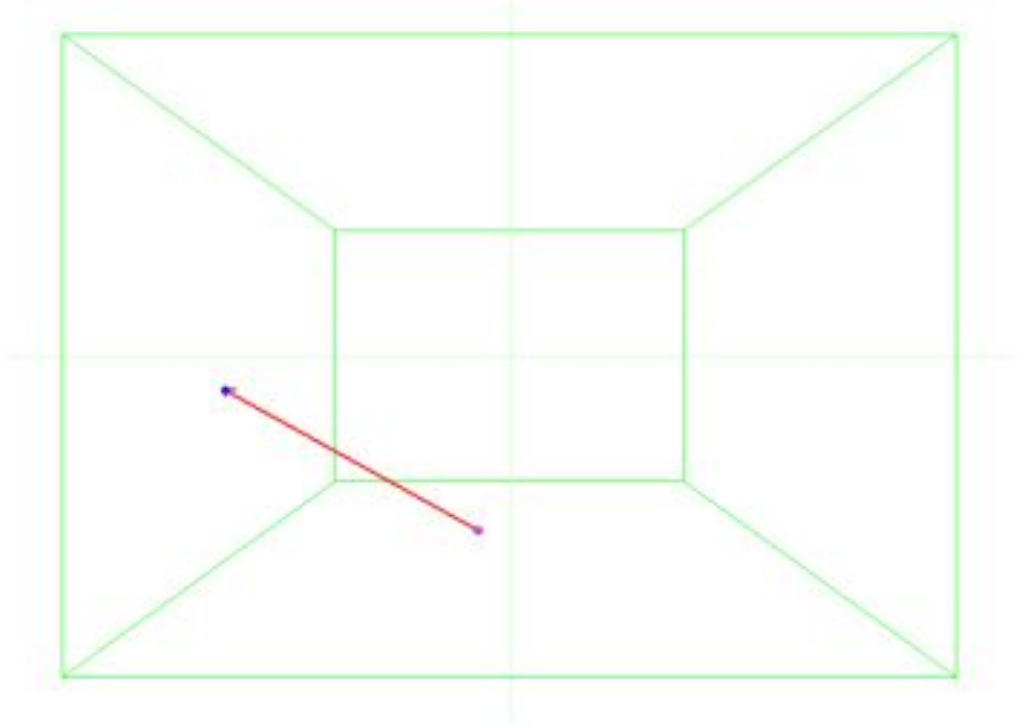


Figure 5.6: Neural network structures for neural evolution for process noise with a gain of 1.5

Table 5.6: Parameter values for the network in Figure 5.6.								
Neuron	Type	x	y	z	T_h	T_{rf}	g	α
0	input	0.342	0.0579	-0.451	20	0.0001	–	–
1	output	0.0427	0.331	-0.356	0.156	0.0001	0.0107	0.608

Table 5.7: Connection values for the network in Figure 5.6.

Parent	Child	Weight
0	1	0.0647
1	1	0.00449
1	0	-1.90

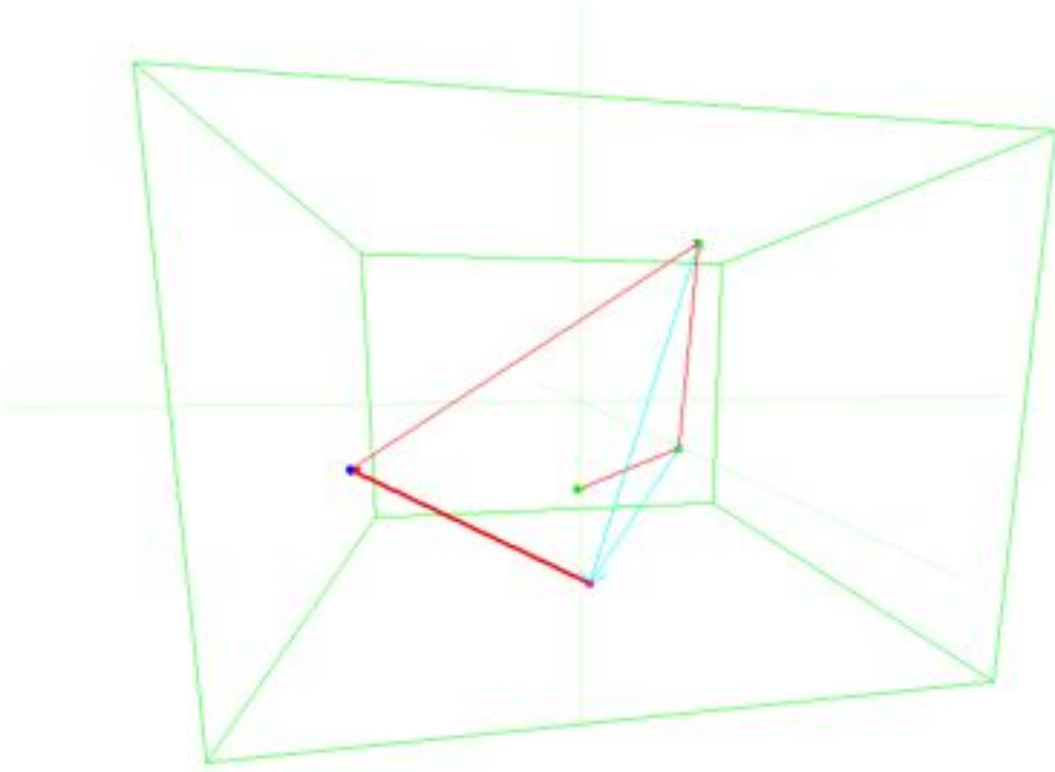


Figure 5.7: A more complex neural network structure developed for neural evolution in the presences of process noise with a gain of 2.0.

Table 5.8: Parameter values for the network in Figure 5.7.

Neuron	Type	x	y	z	T_h	T_{rf}	g	α
0	input	0.342	0.0578	-0.451	2.01	0.0001	—	—
1	output	0.0427	0.331	-0.357	0.281	0.0001	0.0421	0.003
2	hidden	-0.308	-0.487	0.178	0.420	0.570	—	—
3	hidden	-0.324	0.216	0.232	0.429	0.798	—	—
4	hidden	0.026	0.209	-0.134	0.515	0.116	—	—

Table 5.9: Connection values for the network in Figure 5.7.

Parent	Child	Weight
0	2	-0.0284
0	1	0.0647
1	0	-3.47
1	3	0.396
1	1	0.0160
2	1	0.139
3	4	-0.0436
3	2	-0.258
4	4	0.645

5.4 Sumamry

The discrete event network simulator can function as a regulator for the error signal of closed loop continuous time dynamic systems, in this case, the cart-and-pendulum. The nerual evolution algorithm was able to improve the performance of the cart-and-pendulum with a sepearte closed loop feedback controller.

Chapter 6

Conclusions

This work presents a method of modeling and design of neural networks with a discrete event neural simulation for control applications. The discrete event neural simulation focuses on the transmission of events between neurons in a network, rather than modeling the events themselves. This results in computationally efficient simulations for complex neural networks. The behavior and effectiveness of the discrete event neural simulator was evaluated from experiments using two different closed loop systems with continuous time dynamics. A neural evolution algorithm was used to build network structures that perform a specific task to meet a control objective in each of these exploratory activities.

For the harmonic oscillator, the neural simulator was used as a feedback controller. The controller constructed by the neural evolution algorithm in the absence of process noise exhibits poor robustness when the closed loop system is subjected to small amounts of noise. This is expected to be the case for more complex problems because the evolutionary algorithm cannot learn about the system's operation in regions of the state space not excited by the noise process. When the evolution of the network takes place in the presence of a small amount of process noise, a more complicated network structure develops, one that is able to accommodate process noise. This complex structure behaves similarly to central pattern generators by

sustaining activity in the network despite negligible stimulation. This may be an instance of the evolutionary algorithm’s development of an internal model of the process embedded in the network structure. The complex network controller behaves robustly in the presence of nontrivial process noise, suggesting the presence of an internal model.

The complexity and nonlinearity of the cart-and-pendulum system appears to prohibit the development of a neural feedback controller directly via the neural evolution algorithm. The neural evolution algorithm can, however, construct a network for the discrete event neural simulator that operates on the error signal of a deterministic feedback controller. The use of a neural network in this fashion provides better performance in the presence of process noise. This may provide guidance for more complex design problems.

6.1 Future Work

Traditional artificial neural networks have relatively few tunable parameters compared to the discrete event neural simulator. The addition of these parameters add more degrees of freedom, resulting in a much larger space of possible networks. In this study, the neural evolution algorithm maintains as much generality as possible in order to search the space of network structures for a given application. There are several proposed methods that may be useful to enhance the efficiency of this search. Identifying sub-networks that perform a desired task and reusing those structures could improve efficiency. This is the approach taken in [55, 60] using traditional artificial neural networks. The same concept can be applied to individual neurons. Neurons with parameters that perform well in certain situations could be reused. Implementing these search methods would refine the existing search methods in the neural evolution algorithm.

Bibliography

Bibliography

- [1] Newton hpc program – high performance computing. <http://newton.utk.edu>, August 2012. 26
- [2] J.J. Abbot. Design tools for pulse–frequency modulated control systems: Error analysis and limit–cycle prediction. Master’s thesis, The University of Utah, 2001. 14
- [3] J. Á Acosta. Furuta’s pendulum: A conservative nonlinear model for theory and practise. *Mathematical Problems in Engineering*, 2010:1–29, 2010. 23
- [4] C.W. Anderson. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, pages 31–37, April 1989. 16, 22
- [5] P.J. Antsaklis and A.N. Michel. *Linear Systems*. The McGraw–Hill Companies, Inc., 1997. 22, 60
- [6] E.J. Bayly. Spectral analysis of pulse frequency modulation in the nervous systems. *IEEE Trans. on Bio–Medical Engineering*, BME–15(4):257–265, October 1968. 13
- [7] T. Bem, M. Cabelguen, and S. Grillner Ö. Ekeberg. From swimming to walking: A single basic network for two different behaviors. *Biological Cybernetics*, 88:79–90, 2003. 11
- [8] G. Bi. Spatiotemporal specificity of synaptic plasticity: Cellular rules and mechanisms. *Biological Cybernetics*, 87:319–332, 2002. 12

- [9] T.V.P. Bliss and G.L. Collingridge. A synaptic model of memory: Long-term potentiation in the hippocampus. *Nature*, 361:31–39, 1993. 12
- [10] A. Bradshaw and J. Shao. Swing-up control of inverted pendulum systems. *Robotica*, 14:397–405, 1996. 23
- [11] J.T. Buchanan. Neural network simulations of coupled locomotor oscillators in the lamprey spinal chord. *Biological Cybernetics*, 66:367–374, 1992. 11, 12
- [12] G. Buzsáki and A. Draguhn. Neuronal oscillations in cortical networks. *Science*, 304:1926–1929, June 2004. 9, 10
- [13] B.S. Cazzolato and A. Prime. On the dynamics of the furuta pendulum. *Journal of Control Science and Engineering*, pages 1–8, 2011. 23
- [14] A.H. Cohen. Effects of oscillator frequency on phase-locking in the lamprey central pattern generator. *Journal of Neuroscience Methods*, 21:13–125, 1987. 11
- [15] Y.N. Jan C.T. Kuo, L.Y. Jan. Dendrite-specific remodeling of *drosophila* sensory neurons requires matrix metalloproteases, ubiquitin-proteasome, and ecdysone signaling. *Proceedings of the National Academy of Sciences*, 102(42):15230–15235, October 2005. 5
- [16] A. Dekkers and Emile Aarts. Global optimization and simulated annealing. *Mathematical Programming*, 50:367–393, 1991. 19
- [17] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*. John Wiley and Sons, Inc., 2 edition, 2001. 16, 18
- [18] Ö. Ekeberg and S. Grillner. Simulations of neuromuscular control in lamprey swimming. *Philosophical Transactions of the Royal Society B*, 354:895–902, 1999. 11, 12
- [19] I.M. Ethell and E.B. Pasquale. Molecular mechanisms of dendritic spine development and remodeling. *Progress in Neurobiology*, 75:161–205, 2005. 12

- [20] R.D. Fields. Mylenation: An overlooked mechanism of synaptic plasticity. *The Neuroscientist*, 11(6):528–531, 2005. 6
- [21] R. Fitzhugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical Journal*, 1:445–466, 1961. 8
- [22] K. Furuta. Control of pendulum: From super mechano-system to human adaptive mechatronics. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, December 2003. 23
- [23] K. Furuta, M. Yamakita, and S. Kobayashi. Swing up control of inverted pendulum. In *International Conference on Industrial Electronics, Control and Instrumentation*, volume 3, pages 2193–2198, 1991. 23
- [24] F. Gabbiani and C. Koch. Principles of spike train analysis. In C. Koch and I. Segev, editors, *Methods in Neural Modeling*, pages 313–410. The MIT Press, 2 edition, 1998. 6, 7
- [25] K. Graichen, M. Treuer, and M. Zeitz. Swing-up of the double pendulum on a cart by feedforward and feedback control with experimental validation. *Automatica*, 43:63–71, 2007. 23
- [26] D.O. Hebb. *The Organization of Behavior: A Neurophysiological Theory*. John Wiley and Sons, Inc., 1949. 5, 12
- [27] S. Herculano-Houzel. The human brain in numbers: A linearly scaled-up primate brain. *Frontiers in Human Neuroscience*, 3(31), November 2009. 5
- [28] A.L Hodgkin and A.F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544, 1953. 7

- [29] S.J. Huang and C.L. Huang. Control of an inverted pendulum using grey prediction model. *IEEE Transactions on Industry Applications*, 36(2):452–458, 2000. 23
- [30] A.J. Ijspeert, A. Crespi, D. Ryczko, and J Cabelguen. From swimming to walking with a salamander robot driven by a spinal cord model. *Science*, 315:1416–1419, March 2007. 11
- [31] K. Inoue, S. Ma, and C. Jin. Neural oscillator network-based controller for meandering locomotion of snake-like robots. In *International Conference on Robotics and Automation*, pages 5064–5069. IEEE, April 2004. 11
- [32] Q. Jarosz. http://en.wikipedia.org/wiki/File:Neuron_Hand-tuned.svg, July 2012. ix, 6
- [33] F.R. Johnston. Exponentially weighted moving average (ewma) with irregular updating periods. *Journal of Operational Research*, 44(7):711–716. 15
- [34] H. Kahalil. *Nonlinear System*. Prentice-Hall, Inc., 3 edition, 2002. 20, 57
- [35] S.A. Kalwy and R. Smith. Mechanisms of myelin basic protein and proteolipid protein targeting in oligodendrocytes (review). *Molecular Membrane Biology*, 11(2):67–78, 1994. 6
- [36] E. Kandel, J. Schwartz, and T. Jessell. *Principles of Neuroscience*. McGraw-Hill Companies, Inc., 4 edition, 2000. 5, 7, 10
- [37] D. Kleinfeld and H. Sompolinsky. Associative neural network model for the generation of temporal patterns. *Biophysical Journal*, 54:1039–1051, December 1988. 11, 12
- [38] J. Kodjabachian and K. Meyer. Evolution and development of neural controllers for locomotion, gradient-following, and obstacle-avoidance in artificial insects. *IEEE Transactions on Neural Networks*, 9(5), September 1998. 19

- [39] E.B. Kosmatopoulos, M.M. Polycarpou, M.A. Chrisodoulou, and P.A. Ioannou. High-order neural network structures for identification of dynamical systems. *IEEE Transaction of Neural Networks*, 6(2):422–431, 1995. 17
- [40] P. Lau and G. Bi. Synaptic mechanisms of persistent reverberatory activity in neuronal networks. *Proceedings of the National Academy of Science*, 102(29):10333–10338, July 2005. 9, 11
- [41] W.A. Lee and et. al. Dynamic remodeling of dendritic arbors in gabaergic interneurons of adult visual cortex. *PLOS: Biology*, 4(2):271–280, February 2006. 5
- [42] K.H. Lundberg and J.K. Roberge. Classical dual-inverted-pendulum control. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, December 2003. 24
- [43] H. Meier, Z. Farwig, and H. Unbehauen. Discrete computer control of a triple-inverted pendulum. *Optimal Control Applications & Methods*, 11:157–171, 1990. 23, 24
- [44] D.C. Montgomery and G.C. Runger. *Applied Statistics and Probability for Engineers*. John Wiley & Sons, Inc., 3 edition, 2003. 40
- [45] J. Nagumo, S. Arimoto, and S. Yoshizawa. An active pulse transmission line simulating nerve axon. *Proceedings of the Institute of Radio Engineers*, 50(10):2061–2070, October 1962. 8
- [46] K.S. Narendra and K. Parthasarathy. Identification and control of dynamical systems using neural networks. *IEEE Transactions of Neural Networks*, 1(1):4–27, 1990. 17
- [47] K. Pathak, J. Franch, and S.K. Agrawal. Velocity and position control of a wheeled inverted pendulum by partial feedback linearization. *IEEE Transactions on Robotics*, 21(2), June 2005. 24

- [48] J.C. Patra, R.N. Pal, B.N. Chatterji, and G. Panda. Identification of nonlinear dynamical systems using functional link artificial neural networks. *IEEE Transactions on Systems, Man, and Cybernetics–Part B: Cybernetics*, 29(2), April 1999. 17
- [49] T. Pavlidis and E.I. Jury. Analysis of a new class of pulse–frequency modulated feedback systems. *IEEE Trans. on Automatic Control*, pages 35–43, January 1965. 13
- [50] L.C. Phillips. Control of a dual inverted pendulum system using linear–quadratic and h–infinity methods. Master’s thesis, Massachusetts Institute of Technology, 1991. 22, 24
- [51] M. I. Rabinovich, P. Varona, A. I. Selverston, and H. D. I. Abarbanel. Dynamic principles in neuroscience. *Reviews of Modern Physics*, 78:1213–1265, October–December 2006. 5, 8, 9, 10, 12
- [52] J. Åkesson and K.J. Åström. Safe manual control of the Furuta pendulum. In *Proceedings of the 2001 IEEE International Conference on Control Applications*, 2001. 23
- [53] A. Rapoport and W.J. Horvath. The theoretical channel capacity of a single neuron as determined by various coding systems. *Information and Control*, 3:335–350, 1960. 13
- [54] K.J. Åström and K. Furuta. Swinging up a pendulum by energy control. *Automatica*, 36:287–295, 1999. 23, 58
- [55] J. Reisinger, K.O. Stanley, and R. Miikkulainen. Evolving reusable neural modules. In *Genetic and Evolutionary Computation Conference*, pages 69–81, 2004. 16, 17, 19, 71

- [56] N. Saravanan and D.B. Fogel. Evolving neurocontrollers using evolutionary programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 217–222, 1994. 16, 22
- [57] N. Saravanan and D.B. Fogel. Evolving neural control systems. *IEEE Expert*, 10(3):23–27, 1995. 16, 22
- [58] G.M. Shepherd. Introduction to synaptic circuits. In G. M. Shepherd, editor, *The Synaptic Organization of the Brain*, pages 1–38. Oxford University Press, 5 edition, 2004. 5, 7
- [59] M. W. Spong. The swing up control problem for the acrobat. *IEEE Control Systems Magazine*, 15(1):49–55, 1995. 23
- [60] K.O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. 16, 17, 19, 71
- [61] P. Varona, M.I. Rabinovich, A.I. Selverston, and Y.I. Arshavsky. Winnerless competition between sensory neurons generates chaos: A possible mechanism for molluscan hunting behavior. *Chaos*, 12(3):672–677, 2002. 11
- [62] A. Vishwanathan, G. Bi, and H.C. Zeringue. Ring-shaped neuronal networks: A platform to study persistent activity. *Lab Chip*, 11:1081–1088, 2011. 11
- [63] Z. Wang, Y. Chen, and N. Feng. Minimum-time swing-up of a rotary inverted pendulum by iterative impulsive control. In *Proceedings of the 2004 American Control Conference*, June 2004. 23
- [64] N. Wiener. *Nonlinear Problems in Random Theory*. The MIT Press, 1958. 9
- [65] M. Yamakita, M. Iwashiro, Y. Sugahara, and K. Furuta. Robust swing up control of double pendulum. In *Proceedings of the American Control Conference*, volume 1, pages 290–295, 1995. 23, 24

- [66] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, September 1999. 18
- [67] K. Yoshida. Swing-up control of an inverted pendulum by energy-based methods. In *Proceedings of the American Control Conference*, June 1999. 23, 58, 60
- [68] W. Zhong and H. Röck. Energy and passivity based control of the double inverted pendulum on a cart. In *Proceeding of the 2001 IEEE International Conference on Control Applications*, September 2001. 23

Appendix

Appendix A

Source Code

Source code listing are provided for all software components. Note that some fields may include systems dependencies which will require changes for other operating system environments.

A.1 NeuralNetwork.h

```
1  /*
   * File:   NeuralNetwork.h
   * Author: Scott Hansen
   *
   * "A Fox entered the house of an actor and,
   * rummaging through all his properties,
   * came upon a Mask, an admirable imitation of a human head.
   * He placed his paws on it and said, 'What a beautiful head! Yet it is
   * of no value, as it entirely lacks brains.' "
   * -Aesop's Fables
11 *
   * Created on January 5, 2011, 10:19 PM
   */
#ifndef NEURALNETWORK_H
#define NEURALNETWORK_H

#include <math.h>
#include <stdlib.h>
#include <set>
```

```

#include <map>
21 #include <list>
#include <deque>
#include <vector>
#include <string>
#include <sstream>
#include <iostream>
#include <algorithm>
using namespace std;

class NeuralNetwork
31 {
private:

    /**
     * A node in the neural network.
     */
    struct Neuron
    {
        Neuron( double x, double y, double z,
                double t_rf, double threshold,
41         bool is_input, bool is_output );
        /* Mutable Parameters */
        double x, y, z;                // spatial location
        double t_rf;                   // refractory period
        double threshold;              // max accumulator value before firing

        /* State Info */
        double accumulator;            // accumulator value at current_time
        double last_fire_time;        // duh
        bool is_input, is_output;
51         set<Neuron*> parents;        // parent connections
        map<Neuron*, double> children; // children connections and weights
    };

    /**
     * A storage contain for events in the Neural Network.
     */
    struct Pulse
    {
        Pulse( Neuron *p_src, Neuron *p_dst, double mag ) : p_dst(p_dst), mag(mag) {};
61         Pulse() : p_dst( 0 ), mag( 0 ) {};
        Neuron *p_dst;

```

```

        double mag;
    };

public:
    /* Constructors */
    NeuralNetwork( istream &in );
    NeuralNetwork( double x_width = 1, double y_width = 1, double z_width = 1);
    NeuralNetwork( NeuralNetwork &orig );
71 ~NeuralNetwork() { clear(); }

    /* Interface */
    void write( ostream &out );
    void read( istream &in );
    void print( ostream &os = cout );
    void printInterface( ostream &os = cout );
    double advanceSimulation( vector<double> &out );
    void reset();
    void applyInputAt( int i, double t, double mag =1 );
81 double getNextTime();
    int getQueueSize() {return queue.size(); }
    double getInputAccAt( int i ) {return input[i]->accumulator; }
    double getOutputAccAt( int i ) {return output[i]->accumulator; }
    void setInputThrAt( int i, double th ) {input[i]->threshold = th; }
    double getInputThrAt( int i ) {return input[i]->threshold; }
    double getOutputGainAt( int i ) {return output_gain[i]; }
    double getOutputGainAt( Neuron *n ) {return output_gain[ output_idx[ n ] ];}
    void setOutputGainAt( int i, double g ) {output_gain[i] = g; }
    double getOutputAlphaAt( int i ) {return output_alpha[i]; }
91 double getOutputAlphaAt( Neuron *n ) {return output_alpha[ output_idx[ n ] ];}
    void setOutputAlphaAt( int i, double a ){output_alpha[i] = a; }
    double getCurrentTime() {return current_time; }
    double getXWidth() {return x_width; }
    double getYWidth() {return y_width; }
    double getZWidth() {return z_width; }
    void setTimeScale( double t ) {t_sc = t; }
    double getTimeScale() {return t_sc; }

    int getpulselistsize() { return pulse_list.size(); }
101 int getfreelistsize() { return pulse_free.size(); }
    double randf( double a, double b ) { return ((b-a)*((double)rand()/((double)RAND_MAX))+a; }
    double randf() { return (double)rand() / (RAND_MAX); }

private:

```

```

/* Helper Functions */
bool    getEmptyLocation( double &x, double &y, double &z );
Neuron *addNeuron( double x, double y, double z,
                  double t_rf, double threshold,
                  bool is_input = false, bool is_output = false);
111 void    removeNeuron( Neuron *n );
void    addConnection( Neuron *p_src, Neuron *p_dst, double w = 1.0 );
void    removeConnection( Neuron *p_src, Neuron *p_dst );
double  getDistance( Neuron *p_src, Neuron *p_dst );
double  getDelay( Neuron* p_src, Neuron* p_dst );
void    pulseAlloc( Neuron *p_dst, double mag, double t );
void    pulseDealloc( Pulse *p );
void    clear();

/* State Info */
121 double x_width, y_width, z_width; // dimensional of the network
double t_sc;                          // time scaling [delay/dist]
int nc;                               // number of connections
double current_time;
vector<Neuron*> neurons;               // a list of all neurons
vector<Neuron*> input;                 // a list of input neurons
vector<Neuron*> output;                // a list of output neurons
map<Neuron*,int> output_idx;           // indices for the output neurons
vector<double> output_gain;            // gain for each output neuron
vector<double> output_alpha;           // parameters for filtering
131

/* Event Management */
multimap< double, Pulse* > queue; // event queue
list<Pulse> pulse_list;
list<Pulse*> pulse_free;

// These helper classes need VIP access
friend class NeuralNetworkFactory;
friend class GLWidget;
friend class Breeder;
141 };

#endif

```

A.2 NeuralNetwork.cpp

```
#include "NeuralNetwork.h"
```

```

/**
 * Neuron constructor.
 */
NeuralNetwork::Neuron::Neuron( double x, double y, double z,
7      double t_rf, double threshold,
      bool is_input, bool is_output )
: x(x), y(y), z(z), t_rf(t_rf), threshold(threshold),
  accumulator(0.0), last_fire_time(-1),
  is_input(is_input), is_output(is_output)
{}

/**
 * NeuralNetwork constructor. Load a network structure from a file.
 */
17 NeuralNetwork::NeuralNetwork( istream &in )
{
    read( in );
}

/**
 * NeuralNetwork constructor.
 */
NeuralNetwork::NeuralNetwork ( double x_width, double y_width, double z_width )
: x_width(x_width), y_width(y_width), z_width(z_width),
27   current_time(0), t_sc(1), nc(0)
{}

/**
 * Copy constructor.
 */
NeuralNetwork::NeuralNetwork( NeuralNetwork &orig )
{
    stringstream ss;
    orig.write( ss );
37   read( ss );
}

/**
 * Write the network in a condensed format.
 */
void NeuralNetwork::write ( ostream &out )
{

```

```

char d = '␣'; // delimiter

47
// write network info
out << neurons.size() << d;
out << x_width << d;
out << y_width << d;
out << z_width << d;
out << t_sc << endl;

// write info for each neuron
map<Neuron*, int> n_map; // map neurons to indices
57 int i = 0;
for ( vector<Neuron*>::iterator itr = neurons.begin(); itr != neurons.end(); itr++ )
{
    n_map[*itr] = i;
    out << i << d;
    out << (*itr)->x << d;
    out << (*itr)->y << d;
    out << (*itr)->z << d;
    out << (*itr)->t_rf << d;
    out << (*itr)->threshold << d;
67 out << (*itr)->is_input << d;
    out << (*itr)->is_output << d;

    if ( (*itr)->is_output )
    {
        out << output_gain[ output_idx[ *itr ] ] << d;
        out << output_alpha[ output_idx[ *itr ] ] << endl;
    }
    else
    out << endl;
77     i++;
}

// write connections
for ( vector<Neuron*>::iterator itr1 = neurons.begin(); itr1 != neurons.end(); itr1++ )
{
    for ( map<Neuron*,double>::iterator itr2 = (*itr1)->children.begin();
        itr2 != (*itr1)->children.end();
        itr2++ )
    {
87         if ( (*itr2).second )
            {

```

```

        out << n_map[*itr1] << d;          // parent neuron
        out << n_map[(itr2).first] << d; // child neuron
        out << (itr2).second << endl;    // weight
    }
}
}
out << endl;
}
97
/**
 * Read the network from a condensed format.
 */
void NeuralNetwork::read ( istream &in )
{
    // reset the network
    clear();

    // read network structure info
107    int nsize;
    in >> nsize;
    in >> x_width >> y_width >> z_width >> t_sc;

    // read info for each neuron and build them
    map<int,Neuron*> n_map; // map indices to neurons
    int idx, type;
    double x, y, z, trf, thresh, g, a;
    bool is_input, is_output;
    for ( int i = 0; i < nsize; i++ )
117    {
        in >> idx >> x >> y >> z >> trf >> thresh >> is_input >> is_output;

        Neuron *p_n = addNeuron( x, y, z, trf, thresh, is_input, is_output );
        n_map[idx] = p_n;

        if ( p_n->is_output )
        {
            in >> g;
            output_gain[ output_idx[ p_n ] ] = g;
127    in >> a;
            output_alpha[ output_idx[ p_n ] ] = a;
        }

    }
}

```

```

        // read in connection weights
        int src, dst;
        double w;

137     while ( in >> src >> dst >> w )
        {
            if ( w )
                addConnection( n_map[src], n_map[dst], w );
        }

    /**
     * Print the network out in a readable format.
     */
    void NeuralNetwork::print( ostream &os )
    {
147         os << "NeuralNetwork_" << this << "_";
        os << neurons.size() << "_neurons_";
        os << nc << "_synapses_";
        os << endl;
        os << output.size() << "_outputs_" << input.size() << "_inputs_" << endl;
        os << "x_width:" << x_width << "y_width:" << y_width << "z_width:" << z_width << "_";
        os << "t_sc:" << t_sc << endl;

        map<Neuron*,int> n_map;
        int i = 0;
157     for ( vector<Neuron*>::iterator itr = neurons.begin(); itr != neurons.end(); itr++ )
        {
            n_map[*itr] = i;
            os << "->_Neuron" << i << "_" << *itr << endl;
            os << "x:" << (*itr)->x << "y:" << (*itr)->y << "z:" << (*itr)->z << endl;
            os << "t_rf:" << (*itr)->t_rf << "threshold:" << (*itr)->threshold << "_";
            if ( (*itr)->is_input )
                os << "input_" ;
            if ( (*itr)->is_output )
                os << "_output_";
167             if ( (*itr)->is_output )
            {
                os << "gain:" << output_gain[ output_idx[ *itr ] ] << "_";
                os << "alpha:" << output_alpha[ output_idx[ *itr ] ];
            }

            os << endl;
            i++;
        }
    }

```

```

    for ( vector<Neuron*>::iterator itr1 = neurons.begin(); itr1 != neurons.end(); itr1++ )
177     {
        for ( map<Neuron*,double>::iterator itr2 = (*itr1)->children.begin();
            itr2 != (*itr1)->children.end();
            itr2++ )
        {
            os << "neuron" << n_map[*itr1] << "→" << n_map[(*itr2).first] << " ";
            os << "delay:" << getDelay( *itr1, (*itr2).first ) << " ";
            os << "weight:" << (*itr2).second << endl;
        }
    }
187 }

/**
 * Print the interface components of the network, i.e. the input and output neurons.
 */
void NeuralNetwork::printInterface( ostream &os )
{
    os << input.size() << " inputs:" << endl;
    for ( int i = 0; i < input.size(); i++ )
        os << i << " " << input[i] << endl;
197
    os << output.size() << " outputs:" << endl;
    for ( int i = 0; i < output.size(); i++ )
    {
        os << i << " " << output[i] << " " << output_idx[output[i]] << " ";
        os << "gain:" << output_gain[i] << endl;
    }

}

207 /**
 * Intelligently allocate a pulse and add it to the queue.
 */
void NeuralNetwork::pulseAlloc( Neuron *p_dst_in, double mag_in, double t )
{
    // create a new pulse if needed
    if ( !pulse_free.size() )
    {
        pulse_list.push_back( Pulse() );
        pulse_free.push_back( &pulse_list.back() );
217 }

```

```

        // allocate the pulse
        Pulse *p = pulse_free.back();
        pulse_free.pop_back();
        p->p_dst = p_dst_in;
        p->mag = mag_in;

        // queue the pulse
        queue.insert( pair<double,Pulse*>( t, p ) );
227 }

/*
 * Deallocate a pulse when it is no longer being used.
 * Warning: the queue might still be queued.
 */
void NeuralNetwork::pulseDealloc( Pulse *p )
{
    pulse_free.push_back( p );
}
237

/**
 * This is the complicated function.
 * Advance the network simulation to the next time in the queue.
 */
double NeuralNetwork::advanceSimulation( vector<double> &out )
{
    // advance time
    current_time = getNextTime();
    if ( current_time >= 0 ) // the queue is not empty
247 {
        // accumulate coincident events
        pair<multimap<double,Pulse*>::iterator,multimap<double,Pulse*>::iterator> ret;
        map<Neuron*,double> pulse_acc;
        ret = queue.equal_range( current_time ); // all events occuring at this time
        for ( multimap<double,Pulse*>::iterator q_itr = ret.first; q_itr != ret.second; q_itr++ )
        {
            Pulse *p_pulse = q_itr->second;
            Neuron *p_neuron = p_pulse->p_dst;

257 // check to see if this neuron has already recieved an event
            if ( pulse_acc.find( p_neuron ) != pulse_acc.end() )
                pulse_acc[ p_neuron ] += p_pulse->mag;
            else

```

```

        pulse_acc[ p_neuron ] = p_pulse->mag; // this creates the entry in pulse_acc

pulseDealloc( p_pulse );
}

    queue.erase( current_time ); // remove pulses that are no longer needed

267    // update the neurons that recieved a pulse
    for ( map<Neuron*,double>::iterator p_itr = pulse_acc.begin(); p_itr != pulse_acc.end();
        p_itr++ )
    {
        Neuron *p_n = (*p_itr).first;
        p_n->accumulator += (*p_itr).second;

        // check if neuron is in refractory period
        if ( p_n->last_fire_time >= 0 && current_time < p_n->last_fire_time + p_n->t_rf )
        {
277            // refractory period
        }
        else if ( p_n->accumulator <= -p_n->threshold || // fire negative pulse
            p_n->accumulator >= p_n->threshold ) // fire positive pulse
        {
            // check sign
            int sgn = 1;
            if ( p_n->accumulator <= -p_n->threshold )
            sgn = -1;

287            // if this is an output neuron, record it
            if ( p_n->is_output )
            out[output_idx[p_n]] = sgn * output_gain[output_idx[p_n]];

            // queue pulses to be sent to child neurons
            for ( map<Neuron*,double>::iterator c_itr = p_n->children.begin();
                c_itr != p_n->children.end();
                c_itr++ )
            {
                double at = current_time; // arrival time
297                if ( p_n == (*c_itr).first ) // self connection
                    at += p_n->t_rf;
                else
                    at += getDelay( p_n, (*c_itr).first );

                pulseAlloc( (*c_itr).first, sgn*(*c_itr).second, at );
            }

```

```

        // decrement accumulator magnitude
        if ( p_n->accumulator <= -p_n->threshold )
307    p_n->accumulator += p_n->threshold;
        else if ( p_n->accumulator >= p_n->threshold )
            p_n->accumulator -= p_n->threshold;

        // fire again if the accumulator is not depleted sufficiently
        if ( p_n->accumulator <= -p_n->threshold || // excessive negative charge
            p_n->accumulator >= p_n->threshold )    // excessive positive charge
        {
            double at = current_time + p_n->t_rf;
            pulseAlloc( p_n, 0, at );
317    }

        p_n->last_fire_time = current_time;
    } // end if
} // end for
    } // end if ( current_time >= 0 )

    return current_time;
}

327 /**
    * Get the next time from the queue
    */
double NeuralNetwork::getNextTime()
{
    if ( queue.empty() )
        return -1;
    else
        return queue.begin()->first;
}

337 /**
    * Stimulate an input neuron at a particular time with a specified
    * stimulation magnitude.
    */
void NeuralNetwork::applyInputAt( int i, double t, double mag )
{
    if ( t < current_time )
        cerr << "Warning: Input applied at past time, ignoring input." << endl;
    else if ( i >= input.size() )

```

```

347         cerr << "Warning: Invalid input index, ignoring input." << endl;
        else
            pulseAlloc( input[i], mag, t );
    }

    /**
     * Find an empty location not already occupied by a neuron.
     */
    bool NeuralNetwork::getEmptyLocation( double &x, double &y, double &z )
    {
357         int max = 100; // maximum number of samples
        bool is_unique = false;

        while ( max-- && !is_unique)
        {
            is_unique = true;
            x = ( x_width == 0 ? 0 : randf( -x_width/2, x_width/2 ) );
            y = ( y_width == 0 ? 0 : randf( -y_width/2, y_width/2 ) );
            z = ( z_width == 0 ? 0 : randf( -z_width/2, z_width/2 ) );

367         for ( vector<Neuron*>::iterator itr = neurons.begin();
            itr != neurons.end();
            ++itr )
        {
            if( ( (*itr)->x == x && (*itr)->y == y && (*itr)->z == z ) )
            {
                is_unique = false;
                break;
            }
        }
377     }

        return is_unique;
    }

    /**
     * Add a neuron to the network and keep track of its I/O status.
     */
    NeuralNetwork::Neuron *NeuralNetwork::addNeuron( double x, double y, double z,
        double t_rf, double threshold,
387         bool is_input, bool is_output )
    {
        Neuron *neuron = new Neuron( x, y, z, t_rf, threshold, is_input, is_output );

```

```

        neurons.push_back( neuron );

    if ( is_input )
        input.push_back( neuron );

    if ( is_output )
397     {
        output_idx[neuron] = output.size();
        output_gain.push_back( 0.001 );
        output_alpha.push_back( 1.0 );
        output.push_back( neuron );
    }

    return neuron;
}

407 /**
    * Remove a neuron from the network. This also removes all parent and child connections, as well
    * as remove the neuron from queue destinations.
    * Does not remove I/O neurons.
    */
void NeuralNetwork::removeNeuron( Neuron *n )
{
    if ( n->is_input || n->is_output )
        return;

417    // erase child connections from parents
    for ( set<Neuron*>::iterator itr = n->parents.begin(); itr != n->parents.end(); itr++ )
        (*itr)->children.erase( n );

    // erase parent connections from children
    for( map<Neuron*, double>::iterator itr = n->children.begin(); itr != n->children.end(); itr++ )
        (*itr).first->parents.erase( n );

    // delete input references if neccessary
    if ( n->is_input )
427        input.erase( find( input.begin(), input.end(), n ) );

    // remove from the neuron list
    neurons.erase( find( neurons.begin(), neurons.end(), n ) );

    // remove relevent queue events

```

```

        for ( multimap<double,Pulse*>::iterator itr = queue.begin(); itr != queue.end(); itr++ )
            if ( (*itr).second->p_dst == n )
                queue.erase( itr );

437     delete n;
    }

    /**
     * Add a connection safely.
     */
    void NeuralNetwork::addConnection( Neuron *p_src, Neuron *p_dst, double w )
    {
        if ( p_src && p_dst )
        {
447         p_src->children[p_dst] = w;
            p_dst->parents.insert( p_src );
            nc++;
        }
    }

    /**
     * Remove a connection safely.
     */
    void NeuralNetwork::removeConnection( Neuron *p_src, Neuron *p_dst )
457    {
        if ( p_src && p_dst )
        {
            p_dst->parents.erase( p_src );
            p_src->children.erase( p_dst );
            nc--;
        }
    }

    /**
467     * Get the distance between two neurons. This can be defined even if no connection
     * exists between the neurons.
     */
    double NeuralNetwork::getDistance ( Neuron *p_src, Neuron *p_dst )
    {
        return sqrt (
            ( p_src->x - p_dst->x ) * ( p_src->x - p_dst->x ) +
            ( p_src->y - p_dst->y ) * ( p_src->y - p_dst->y ) +
            ( p_src->z - p_dst->z ) * ( p_src->z - p_dst->z )

```

```

        );
477 }

/**
 * Get the transmission delay between two neurons. This is simply the distace
 * scaled by the time scaling of the network.
 */
double NeuralNetwork::getDelay ( Neuron *p_src, Neuron *p_dst )
{
    return ( t_sc * getDistance ( p_src, p_dst ) );
}

487
/**
 * Destruct the network.
 */
void NeuralNetwork::clear()
{
    // reset connections
    nc = 0;

    // clear out neuron references
497    input.clear();
    output.clear();
    output_idx.clear();

    // delete neuron pointers
    for ( vector<Neuron*>::iterator itr = neurons.begin(); itr != neurons.end(); itr++ )
        delete (*itr);
    neurons.clear();

    // delete events
507    pulse_free.clear();
    pulse_list.clear();
    queue.clear();

    // reset the time
    current_time = 0.0;
}

/**
 * Clear all event-related content from the system and set the time back to 0.
517 * Prepares the network for a new simulation.
 */

```

```

void NeuralNetwork::reset()
{
    // reset neuron accumulators
    for( vector<Neuron*>::iterator itr = neurons.begin(); itr != neurons.end(); itr++ )
    {
        (*itr)->accumulator = 0;
        (*itr)->last_fire_time = 0;
    }

527    // delete old events
    pulse_free.clear();
    pulse_list.clear();
    queue.clear();

    // reset the time
    current_time = 0.0;
}

```

A.3 NeuralNetworkFactory.h

```

/**
 * Author: Scott Hansen
 * File: NeuralNetworkFactory.h
 * Date:
5  *
 * Helpful utility for creating networks.
 */

#ifndef NEURALNETWORKFACTORY_H
#define NEURALNETWORKFACTORY_H

#include "NeuralNetwork.h"

class NeuralNetworkFactory
15 {
public:
    NeuralNetwork *createNetwork( int i,
        int ninput = 1, int nhidden = 1, int noutput = 1,
        double d = 1 );

private:
    double density( double x, double d ) { return exp(-x*x/d); }
    NeuralNetwork *createNetwork0(); // single neuron

```

```

    NeuralNetwork *createNetwork1(); // two connected neurons
25 NeuralNetwork *createNetwork2();
    NeuralNetwork *createNetwork3();
    NeuralNetwork *createNetwork4( int ninput, int noutput ); // 2 in, 1 out, 0 connections
    NeuralNetwork *createNetwork5( int ninput, int noutput ); // 4 in, 1 out, 0 connections
    NeuralNetwork *createNetwork6(); // same as 5, no connections
    NeuralNetwork *createNetwork7(); // two PCM demodulators
    NeuralNetwork *createNetwork8(); // four PCM demodulators
    NeuralNetwork *createRandomNetwork( int ninput, int nhidden, int noutput,
        double d,
        double xwidth = 1, double ywidth = 1, double zwidth = 1 );
35 };

#endif

```

A.4 NeuralNetworkFactory.cpp

```

1 #include "NeuralNetworkFactory.h"

/**
 * Create the network specified by the index i. Additional inputs are only used when
 * creating a random network.
 */
NeuralNetwork *NeuralNetworkFactory::createNetwork( int i,
    int ninput, int nhidden, int noutput,
    double d )
{
11 NeuralNetwork *n;

    switch ( i )
    {
        case 0:
            n = createNetwork0();
            break;
        case 1:
            n = createNetwork1();
            break;
21 case 2:
            n = createNetwork2();
            break;
        case 3:

```

```

        n = createNetwork3();
        break;
    case 4:
        n = createNetwork4( ninput, noutput );
        break;
    case 5:
31     n = createNetwork5( ninput, noutput );
        break;
    case 6:
        n = createNetwork6();
        break;
    case 7:
        n = createNetwork7();
        break;
    case 8:
        n = createNetwork8();
41     break;
    default:
        n = createRandomNetwork( ninput, nhidden, noutput, d );
    }

    return n;
}

/**
 * Single neuron.
51 */
NeuralNetwork *NeuralNetworkFactory::createNetwork0()
{
    NeuralNetwork *n = new NeuralNetwork();
    NeuralNetwork::Neuron *n1;

    n1 = n->addNeuron( -0.0, 0.0, 0.0, 0.0001, 10, true, true );

    return n;
}

61 NeuralNetwork *NeuralNetworkFactory::createNetwork1()
{
    NeuralNetwork *n = new NeuralNetwork();
    NeuralNetwork::Neuron *neuron1, *neuron2;

    // parameters

```

```

double g1 = 1;
double a1 = 0.1;
double th = 20;
71 double ts = 1;

neuron1 = n->addNeuron( -0.5, 0.0, 0.0, 0.0001, th, true, false );
neuron2 = n->addNeuron( 0.5, 0.0, 0.0, 0.0001, 1.0, false, true );

n->addConnection( neuron1, neuron2, 1.0 );
n->addConnection( neuron2, neuron1, -1.0 );
n->setOutputGainAt( 0, g1 );
n->setOutputAlphaAt( 0, a1 );
n->setTimeScale( ts );
81
return n;
}

NeuralNetwork *NeuralNetworkFactory::createNetwork2()
{
    NeuralNetwork *n = new NeuralNetwork();
    NeuralNetwork::Neuron *input, *hidden1, *hidden2, *output1, *output2;
    double t_rf = 0.0001;

91 input = n->addNeuron( -0.5, 0.0, 0.0, t_rf, 1.0, true, false );
hidden1 = n->addNeuron( 0.0, 0.5, 0.0, t_rf, 1.0 );
hidden2 = n->addNeuron( 0.0, -0.5, 0.0, t_rf, 1.0 );
output1 = n->addNeuron( 0.5, 0.1, 0.0, t_rf, 1.0, false, true );
output2 = n->addNeuron( 0.5, -0.1, 0.0, t_rf, 1.0, false, true );

n->addConnection( input, hidden1, 1.0 );
n->addConnection( input, hidden2, 1.0 );
n->addConnection( hidden1, output1, 1.0 );
n->addConnection( hidden2, output2, 1.0 );
101 n->addConnection( output1, hidden2, 1.0 );
n->addConnection( output2, hidden1, 1.0 );

return n;
}

NeuralNetwork *NeuralNetworkFactory::createNetwork3()
{
    NeuralNetwork *n = new NeuralNetwork();
    NeuralNetwork::Neuron *n1, *n2;

```

```

111     double t_rf = 0.0001;

        // parameters
        double g1 = 1;
        double th = 17;
        double ts = 1;

        n1 = n->addNeuron( 0.5, 0.1, 0.0, t_rf, th, true, true );
        n2 = n->addNeuron( -0.5, 0.1, 0.0, t_rf, 1, false, true );

121     n->addConnection( n1, n2, 1.0 );
        n->output_gain[0] = g1;
        n->output_gain[1] = g1;
        n->setTimeScale( ts );

        return n;
    }

    NeuralNetwork *NeuralNetworkFactory::createNetwork4( int ninput, int noutput )
    {
131         NeuralNetwork *n = new NeuralNetwork();
        NeuralNetwork::Neuron *n1, *n2, *n3;

        double t_rf = 0.0001;
        double g = 1;
        double th = 10;
        double ts = 1;

        n1 = n->addNeuron( 0.5, 0.1, 0.0, t_rf, th, true, false ); // input1
        n2 = n->addNeuron( 0.5, -0.1, 0.0, t_rf, th, true, false ); // input2
141     n3 = n->addNeuron( -0.5, 0.1, 0.0, t_rf, th, false, true ); // output 1

        n->setOutputGainAt( 0, g );
        n->setTimeScale( ts );

        return n;
    }

    /**
     * Ditto, except with for inputs
151     */
    NeuralNetwork *NeuralNetworkFactory::createNetwork5( int ninput, int noutput )
    {

```

```

NeuralNetwork *n = new NeuralNetwork();
NeuralNetwork::Neuron *n1, *n2, *n3, *n4, *n5;

double t_rf = 0.0001;
double g = 1;
double th = 10000000;
double ts = 1;

161
n1 = n->addNeuron( 0.5, 0.1, 0.0, t_rf, th, true, false ); // input1
n2 = n->addNeuron( 0.5, -0.1, 0.0, t_rf, th, true, false); // input2
n3 = n->addNeuron( 0.5, 0.2, 0.0, t_rf, th, true, false ); // input1
n4 = n->addNeuron( 0.5, -0.2, 0.0, t_rf, th, true, false); // input2
n5 = n->addNeuron( -0.5, 0.1, 0.0, t_rf, th, false, true ); // output 1

n->addConnection( n1, n5, 1 );
n->addConnection( n2, n5, -1 );
n->addConnection( n3, n5, 1 );
171 n->addConnection( n4, n5, -1 );

n->setOutputGainAt( 0, g );
n->setTimeScale( ts );

return n;
}

/**
 * Same as network 5. No preset connections.
181 */
NeuralNetwork *NeuralNetworkFactory::createNetwork6()
{
    NeuralNetwork *n = new NeuralNetwork();
    NeuralNetwork::Neuron *n1, *n2;
    double t_rf = 0.0001;

    // parameters
    double g1 = 1;
    double th = 17;
191 double ts = 1;

    n1 = n->addNeuron( 0.5, 0.1, 0.0, t_rf, th, true, true );
    n2 = n->addNeuron( -0.5, 0.1, 0.0, t_rf, 1, false, true );

    n->output_gain[0] = g1;

```

```

        n->output_gain[1] = g1;

        n->setTimeScale( ts );

201     return n;
    }

    /**
     * Two state derivative estimation.
     */
    NeuralNetwork *NeuralNetworkFactory::createNetwork7()
    {
        NeuralNetwork *n = new NeuralNetwork();
        NeuralNetwork::Neuron *n1, *n2, *n3, *n4;
211     double t_rf = 0.0001;

        // parameters
        double g1 = 1;
        double th = 17;
        double ts = 1;

        n1 = n->addNeuron( 0.5, 0.1, 0.0, t_rf, th, true, true );
        n2 = n->addNeuron( -0.5, 0.1, 0.0, t_rf, 1, false, true );
        n3 = n->addNeuron( 0.5, -0.1, 0.0, t_rf, th, true, true );
221     n4 = n->addNeuron( -0.5, -0.1, 0.0, t_rf, 1, false, true );

        n->output_gain[0] = g1;
        n->output_gain[1] = g1;
        n->output_gain[2] = g1;
        n->output_gain[3] = g1;

        n->setTimeScale( ts );

        return n;
231 }

    /**
     * Four state derivative estimation.
     */
    NeuralNetwork *NeuralNetworkFactory::createNetwork8()
    {
        NeuralNetwork *n = new NeuralNetwork();
        NeuralNetwork::Neuron *n1, *n2, *n3, *n4,

```

```

        *n5, *n6, *n7, *n8;

241 double t_rf = 0.0001;

    // parameters
    double g1 = 0.0001;
    double th = 17;
    double ts = 1;

    n1 = n->addNeuron( 0.5, 0.1, 0.0, t_rf, th, true, true );
    n2 = n->addNeuron( -0.5, 0.1, 0.0, t_rf, 1, false, true );
    n3 = n->addNeuron( 0.5, -0.1, 0.0, t_rf, th, true, true );
251 n4 = n->addNeuron( -0.5, -0.1, 0.0, t_rf, 1, false, true );
    n5 = n->addNeuron( 0.5, 0.3, 0.0, t_rf, th, true, true );
    n6 = n->addNeuron( -0.5, 0.3, 0.0, t_rf, 1, false, true );
    n7 = n->addNeuron( 0.5, -0.3, 0.0, t_rf, th, true, true );
    n8 = n->addNeuron( -0.5, -0.3, 0.0, t_rf, 1, false, true );

    n->output_gain[0] = g1;
    n->output_gain[1] = g1;
    n->output_gain[2] = g1;
    n->output_gain[3] = g1;
261 n->output_gain[4] = g1;
    n->output_gain[5] = g1;
    n->output_gain[6] = g1;
    n->output_gain[7] = g1;

    n->setTimeScale( ts );

    return n;
}

271 NeuralNetwork *NeuralNetworkFactory::createRandomNetwork(int ninput, int nhidden, int noutput,
        double d,
        double xwidth, double ywidth, double zwidth)
{
    NeuralNetwork *n = new NeuralNetwork( xwidth, ywidth, zwidth );
    vector<NeuralNetwork::Neuron*> neurons;
    NeuralNetwork::Neuron *neuron;
    vector<NeuralNetwork::Neuron*>::iterator itr1, itr2;
    double t_rf = 0.0001;
    double g = n->randf();
281 double ts = n->randf();
    double x, y, z;

```

```

// create input neurons
for ( int i = 0; i < ninput; i++ )
{
    if ( n->getEmptyLocation( x, y, z ) )
    {
        neuron = n->addNeuron( x, y, z,
                                t_rf,
291         n->randf(), // threshold
                                true, false );
        neurons.push_back( neuron );
    }
    else
    {
        cerr << "Error: could not find location for input!" << endl;
    }
}

301 // create hidden neurons
for ( int i = 0; i < nhidden; i++ )
{
    if ( n->getEmptyLocation( x, y, z ) )
    {
        neuron = n->addNeuron( x, y, z,
                                t_rf,
                                n->randf(), // threshold
                                false, false );
        neurons.push_back( neuron );
311 }
    else
    {
        cerr << "Error: could not find location for hidden!" << endl;
    }
}

// create output neurons
for ( int i = 0; i < noutput; i++ )
{
321     if ( n->getEmptyLocation( x, y, z ) )
    {
        neuron = n->addNeuron( x, y, z,
                                t_rf,
                                n->randf(), // threshold

```

```

        false, true );
    neurons.push_back( neuron );
}

else
{
331     cerr << "Error: could not find location for output!" << endl;
}

}

// connect neurons
if ( d > 0 )
{
    for ( itr1 = neurons.begin(); itr1 != neurons.end(); itr1++ )
    {
        for ( itr2 = neurons.begin(); itr2 != neurons.end(); itr2++ )
341     {
            if ( n->randf() < d )
                n->addConnection( *itr1, *itr2, pow(-1,rand()%2)*n->randf() );
        }
    }
}

// randomize output gain
for ( vector<double>::iterator itr = n->output_gain.begin();
351   itr != n->output_gain.end();
    itr++ )
    (*itr) = g;

// randomize time scale
n->setTimeScale( ts );

return n;
}

```

A.5 Breeder.h

1

```

/*
 * File:   Breeder.h
 * Author: Scott Hansen
 *

```

```

    * This class handles inter-generational modifications to the network, including random
    * mutations as well as network cross over.
    *
    * Created on January 5, 2011, 10:19 PM
11 */
#ifndef BREEDER_H
#define BREEDER_H

#include <map>
using namespace std;

#include "NeuralNetwork.h"

class Breeder
21 {
public:
    /* Mutation Types */
    enum mutation_t{ WEIGHT, THRESHOLD, REFRACTORY_PERIOD, TIME_SCALE, GAIN,
        MOVE, REMOVE_CONNECTION, ADD_CONNECTION, ADD_NEURON, REMOVE_NEURON, RANDOM };

    /* Constructors */
    Breeder( NeuralNetwork *n );
    ~Breeder();

31 /* Interface */
    int mutate( mutation_t m = RANDOM );
    int mutate_parameter();
    int mutate_structure();
    int breed( NeuralNetwork *n1, NeuralNetwork *n2 );

private:
    /* Random Mutations */
    int mutateW();
    int mutateTh();
41 int mutateRf();
    int mutateTs();
    int mutateG();
    int mutateA();
    int mutateMv();
    int mutateRmConn();
    int mutateAddConn();
    int mutateAddN();
    int mutateRmN();

```

```

        NeuralNetwork *n;
51     int nmut;
    };

#endif

```

A.6 Breeder.cpp

```

#include "Breeder.h"

/* Constructors */
5 Breeder::Breeder( NeuralNetwork* n )
    : n(n), nmut(10)
{}

Breeder::~Breeder()
{}

/**
 * Apply a mutation to the network.
 *
15 */
int Breeder::mutate( mutation_t m )
{
    int result = 0;

    if ( !n->neurons.size() ) // the network is empty, so add a neuron
    {
        // cout << " an";
        result = mutateAddN();
    }
25 else if ( n->nc == 0 )
    {
        // cout << "ac ";
        result = mutateAddConn();
    }
    else
    {
        mutation_t mut;
        if ( m != RANDOM )
            mut = m;
35     else // apply a random mutation

```

```

mut = mutation_t( rand()%nmut );

    switch ( mut )
    {
    case WEIGHT:
        result = mutateW();
        break;
    case THRESHOLD:
        result = mutateTh();
45     break;
    case REFRACTORY_PERIOD:
        result = mutateRf();
        break;
    case TIME_SCALE:
        result = mutateTs();
        break;
    case GAIN:
        result = mutateG();
        break;
55     case MOVE:
        result = mutateMv();
        break;
    case REMOVE_CONNECTION:
        result = mutateRmConn();
        break;
    case ADD_CONNECTION:
        if ( n->neurons.size() < 2 )
            result = mutateAddN();
        else
65         result = mutateAddConn();
        break;
    case ADD_NEURON:
        result = mutateAddN();
        break;
    case REMOVE_NEURON:
        result = mutateRmN();
        break;
    default:
        cerr << "Warning: invalid mutation (m)" << endl;
75     }
    }

    return result;
}

```

```

/**
 * Mutate a random parameter.
 */
int Breeder::mutate_parameter()
{
85   int m = 7;
      int result = 0;
      int mut = rand() % m;

      switch ( mut )
      {
      case 0:
          result = mutateW();
          break;
      case 1:
95         result = mutateTh();
          break;
      case 2:
          result = mutateRf();
          break;
      case 3:
          result = mutateTs();
          break;
      case 4:
          result = mutateG();
105         break;
      case 5:
          result = mutateA();
          break;
      case 6:
          result = mutateMv();
          break;
      default:
          cerr << "Warning: invalid mutation(p)" << endl;
      }
115   return result;
}

/**
 * Mutate the structure of the network (perhaps catastrophically).
 */

```

```

int Breeder::mutate_structure()
{
    int m = 4;
125    int result = 0;

    if ( !n->neurons.size() ) // the network is empty, so add a neuron
        result = mutateAddN();
    else if ( n->nc == 0 )
        result = mutateAddConn();
    else
    {
        int mut = rand() % m;
        switch ( mut )
135    {
        case 0:
            result = mutateRmConn();
            break;
        case 1:
            if ( n->neurons.size() < 2 )
                result = mutateAddN();
            else
                result = mutateAddConn();
            break;
145    case 2:
            result = mutateAddN();
            break;
        case 3:
            result = mutateRmN();
            break;
        default:
            cerr << "Warning: invalid mutation(s)" << endl;
        }
    }

155    return result;
}

/**
 * Modify the weigh of a random neuron in the network. This increments
 * or decrements the existing value by a small percentage.
 */
int Breeder::mutateW()
{

```

```

165   NeuralNetwork::Neuron *p_n = n->neurons[rand()%n->neurons.size()]; // pick random neuron
      double delta = n->randf(); // the percent change to make

      if ( p_n->children.size() ) // change weight to child
      {
          int nrand = rand()%p_n->children.size(); // pick a random child
          int i = 0;
          for( map<NeuralNetwork::Neuron*,double>::iterator itr = p_n->children.begin();
              itr != p_n->children.end();
              ++itr, ++i )
175   {
          if ( i == nrand )
          {
              double w_old = itr->second;
              double w_new = w_old + pow(-1,rand()%2) * delta * w_old;

              if ( w_new )
                  p_n->children[itr->first] = w_new;
              else
                  n->removeConnection( p_n, itr->first );
185
              break;
          }
      }

      else if ( p_n->parents.size() ) // change weight from parent
      {
          int nrand = rand()%p_n->parents.size(); // pick a random parent
          int i = 0;
          for ( set<NeuralNetwork::Neuron*>::iterator itr = p_n->parents.begin();
              itr != p_n->parents.end();
195   ++itr, ++i )
          {
              if ( i == nrand )
              {
                  double w_old = (*itr)->children[p_n];
                  double w_new = w_old + pow(-1,rand()%2) * delta * w_old;

                  if ( w_new )
                      (*itr)->children[p_n] = w_new;
205   else
                      n->removeConnection( (*itr), p_n );

```

```

        break;
    }
}
}
else
    return -1;

215     return 0;
}

/**
 * Change the threshold of a random neruon in the network by some percentage
 */
int Breeder::mutateTh()
{
    NeuralNetwork::Neuron *p_n = n->neurons[rand()%n->neurons.size()]; // random neuron
    double delta = n->randf();
225     double t_old = p_n->threshold;
    double t_new = 0;

    while ( !t_new )
        t_new = t_old + pow(-1,rand()%2) * delta * t_old;

    p_n->threshold = t_new;

    return 0;
}

235
/**
 * Change the refractory period of a neuron by some percentage.
 */
int Breeder::mutateRf()
{
    NeuralNetwork::Neuron *p_n = n->neurons[rand()%n->neurons.size()]; // random neuron
    double delta = n->randf();
    double t_old = p_n->t_rf;
    double t_new = 0;
245

    while ( !t_new )
        t_new = t_old + pow(-1,rand()%2) * delta * t_old;

    return 0;
}

```

```

/**
 * Randomly change the time scale of the system by a percentage.
 */
255 int Breeder::mutateTs()
{
    double delta = n->randf();
    double old = n->t_sc;
    double new_ts = 0;

    while( !new_ts )
        new_ts = old + pow(-1,rand()%2) * delta * old;

    n->t_sc = new_ts;
265
    return 0;
}

/**
 * Randomly change the gain on the network by a percentage
 */
int Breeder::mutateG()
{
275     int idx = rand()%n->output.size(); // random output neuron
    double delta = n->randf();
    double old = n->output_gain[idx];
    n->output_gain[idx] = old + pow(-1,rand()%2) * delta * old;
    return 0;
}

/**
 * Randomly change the output filter parameter.
 */
285 int Breeder::mutateA()
{
    int idx = rand()%n->output.size(); // random output neuron
    double delta = n->randf();
    double old = n->output_alpha[idx];
    n->output_alpha[idx] = old + pow(-1,rand()%2) * delta * old;
    return 0;
}

```

```

295  /**
    * Move a random Neuron to a random (empty) location.
    */
    int Breeder::mutateMv()
    {
        NeuralNetwork::Neuron *p_n = n->neurons[rand()%n->neurons.size()]; // random neuron
        double x, y, z;
        bool is_unique;

        is_unique = n->getEmptyLocation( x, y, z );

305
        if ( is_unique )
        {
            p_n->x = x;
            p_n->y = y;
            p_n->z = z;
        }
        else
            return -2;
    }

315
    /**
    * Remove a random connection from a random neuron in the network.
    */
    int Breeder::mutateRmConn()
    {
        NeuralNetwork::Neuron *p_n;

        // select a neuron that actually has an outgoing connection
        vector<int> idxs; // list of possible neuron indices
325     for ( int i = 0; i < n->neurons.size(); i++ )
            idxs.push_back( i );
        random_shuffle( idxs.begin(), idxs.end() );

        int idx1; // parent neuron
        for ( idx1 = 0; idx1 < idxs.size(); idx1++ )
        {
            p_n = n->neurons[ idxs[ idx1 ] ];
            if ( p_n->children.size() ) // found a neuron with children
                break;
335     }

```

```

    if ( idx1 == idxs.size() ) // no n->neurons have outgoing connections
        return -3;

    // pick a random connection
    int to_remove = rand()%p_n->children.size();
    int count = 0;

    for ( map<NeuralNetwork::Neuron*,double>::iterator itr = p_n->children.begin();
345   itr != p_n->children.end();
        itr++ )
    {
        if ( to_remove == count )
        {
            n->removeConnection( p_n, (*itr).first );
            break;
        }
        else
            count++;
355     }

    return 0;
}

/**
 * Add a random connection to a random neuron in the network.
 */
int Breeder::mutateAddConn()
{
365   NeuralNetwork::Neuron *p_n1, *p_n2; // neurons to add a connection between

    // select a neuron that actually has an outgoing connection
    vector<int> idxs; // list of possible neuron indices from neurons
    for ( int i = 0; i < n->neurons.size(); i++ )
        idxs.push_back( i );
    random_shuffle( idxs.begin(), idxs.end() );

    int idx1 = 0; // the index of the first neuron
    p_n1 = n->neurons[ idxs[idx1] ];
375

    // select a neuron that is not already connected to p_n1
    random_shuffle( idxs.begin(), idxs.end() );
    for ( int idx2 = 0; idx2 < idxs.size(); idx2++ )
    {

```

```

        p_n2 = n->neurons[ idxs[ idx2 ] ];
        if ( p_n1->children.find( p_n2 ) == p_n1->children.end() ) // no connection exists
    {
        // connect p_n1 to p_n2
        double r = 0;
385    while ( !r )
            r = pow(-1,rand()%2)*n->randf();

        n->addConnection( p_n1, p_n2, r );
        return 0;
    }

    // else no connection was made
    return -4;
395 }

/**
 * Create a neuron with random values and place it in a random (empty)
 * location on the grid. Then connect it randomly to the other neurons.
 */
int Breeder::mutateAddN()
{
    double x, y, z;
    bool is_unique;
405    NeuralNetwork::Neuron *p = 0, *m = 0, *c = 0;

    is_unique = n->getEmptyLocation( x, y, z );
    if ( is_unique )
    {
        p = n->neurons[ rand()%n->neurons.size() ]; // parent
        c = n->neurons[ rand()%n->neurons.size() ]; // child
        m = n->addNeuron( x, y, z,
            n->randf(),
            n->randf() );
415

        // add connections
        double r = 0;
        while ( !r )
            r = pow(-1,rand()%2)*n->randf();
        n->addConnection( p, m, r );

        r = 0;

```

```

        while ( !r )
            r = pow(-1,rand()%2)*n->randf();
425     n->addConnection( m, c, r );
    }
    else
        return -5;

    return 0;
}

int Breeder::mutateRmN()
{
435     vector<int> idxs; // list of possible neuron indices from neurons
    for ( int i = 0; i < n->neurons.size(); i++ )
        idxs.push_back( i );
    random_shuffle( idxs.begin(), idxs.end() );

    for ( int i = 0; i < idxs.size(); i++ )
    {
        if ( n->neurons[i]->is_input || n->neurons[i]->is_output )
            continue;
        else
445     {
            n->removeNeuron( n->neurons[i] );
            return 0;
        }
    }

    // could not delete a neuron
    return -6;
}

```

A.7 sim_harmonic.h

```

/**
 * Author: Scott Hansen
 * File: sim_harmonic.h
 * Date: Feb 16, 2012
 *
 * Contains simulation specifications for working with the harmonic oscillator.
7 * Values presented are the default values. Values can be change via the command line
 * using the swiches specified in process_sim_arguments().
 * Note: these variables and functions are globally defined since DLSODE doesn't

```

```

*           play nice with object oriented code --
*/

#ifndef SIM_H
#define SIM_H

#include <string>
17 #include <string.h>
#include <math.h>
#include <time.h>
#include <cstdlib>
#include <sstream>
#include <iostream>
#include <fstream>
#include <ostream>
#include <queue>
#include <deque>
27 using namespace std;

/* Debugging */
// #define DEBUG
// #define DEBUG_NOISE
// #define DEBUG_STATS

/* System Parameters */
double m = 0.1; // pendulum mass (Kg)
double k = 1; // friction coefficient
37 double pi = 3.14159265358979;
#define SIMFAIL 100000000.0

/* I/O */
string netfile;

/* Neuralnetwork Properties */
int seed = time(NULL); // random seed
double density = 0; // connection density
double xsize = 1; // network x dimension size
47 double ysize = 1; // network y dimension size
double zsize = 1; // network z dimension size
int ninput = 2; // 2number of input neurons
int nhidden = 1; // 1number of hidden neurons
int noutput = 1; // 4number of output neurons
int nidx = 4; // 7default network

```

```

/* EP Variables */
double      fitness_tol  = 0.009;
int          gen_max     = 500;
57 double    state_tol[] = { 2.0, 0.0 };
double      T           = 0.99; // SA temperature
double      fail_tol    = 0.3;
double      q_tol       = 500;

/* DLSODAR Simulation Variables */
int          neq         = 2;    // number of first order diff eq's and the input to the system
int          ng          = 0;    // number of boundary condition equations
double      dtout        = 0.01; // sampling time
int          itol        = 1;    // 1: scalar rep. for atol, 2: array rep. for each state
67 double    rtol        = 0.0;  // relative tolerance
double      abtol        = 0.00001; // absolute tolerance
int          itask       = 1;    // 1: normal task specification
int          iopt        = 0;    // 0: no optional inputs, 1: optional inputs specified
int          lrw         = 32 + neq*(neq+9) + 3*ng; // length of real work array
int          liw         = 20 + neq; // length of integer work array
double      endtime      = 30;   // endtime
int          jt          = 2;    // 1: user supplied jac 2: internally generated jac
double      U            = 0;    // input
double      ic[]         = {1.0, 0.0}; // initial conditions
77

/* Windowing Variables */

/* Noise Model */
bool pnoise = false;
bool mnoise = false;
vector<double> proc_noise( ninput, 0 ); // one for each input
vector<double> meas_noise( neq, 0 ); // one for each state
double p_alpha = 0.01;
double m_alpha = 0.01;
87

/* Function Implementation */
void process_sim_arguments( int argc, char** argv )
{
    for ( int i = 1; i < argc; i++ )
    {
        if ( strcmp( argv[i], "-h" ) == 0 )
        {
            cerr << "just read the switches in the code, its not difficult..." << endl;

```

```

        exit(0);
97     }

        else if ( strcmp( argv[i], "-seed"          ) == 0 ) // network properties
seed = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-density"       ) == 0 )
density = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-xsize"        ) == 0 )
xsize = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-ysize"        ) == 0 )
ysize = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-zsize"        ) == 0 )
107    zsize = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-ninput"       ) == 0 )
ninput = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-nhidden"      ) == 0 )
nhidden = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-noutput"     ) == 0 )
noutput = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-nidx"        ) == 0 )
nidx = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-fitness_tol"   ) == 0 ) // EP variables
117    fitness_tol = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-statetol0"    ) == 0 )
state_tol[0] = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-statetol1"    ) == 0 )
state_tol[1] = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-genmax"       ) == 0 )
gen_max = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-dtout"        ) == 0 ) // DLSODAR variables
dtout = atof( argv[i] );
        else if ( strcmp( argv[i], "-rtol"        ) == 0 )
127    rtol = atof( argv[i] );
        else if ( strcmp( argv[i], "-abtol"       ) == 0 )
abtol = atof( argv[i] );
        else if ( strcmp( argv[i], "-endtime"     ) == 0 )
endtime = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-ic0"        ) == 0 )
ic[0] = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-ic1"        ) == 0 )
ic[1] = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-pnoise"      ) == 0 ) // noise variables
137    pnoise = true;
        else if ( strcmp( argv[i], "-mnoise"     ) == 0 )

```

```

        mnoise = true;
        else if ( strcmp( argv[i], "-palpha"      ) == 0 )
            p_alpha = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-malpha"      ) == 0 )
            m_alpha = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-d"          ) == 0 ) // other
            netfile = argv[i+1];
    }
147 }

/* DLSODAR Functions */
void F( int &NEQ, double &T , double *X , double *XDOT )
{
    XDOT[0] = X[1];
    XDOT[1] = -k/m*X[0] + U/m;
}

void JAC( int &NEQ, double &T, double *X, int &ML, int &MU, double *PD, int &NROWPD )
157 {
}

void G( int &NEQ, double &T, double *X, int &NG, double *GOUT )
{
}

/* Function Declarations */
extern "C"
{
167 void dlsodar_( void (*F)(int&, double&, double*, double*),
                int &NEQ, double *X, double &T, double &TOUT, int &ITOL, double &RTOL,
                double &ATOL, int &ITASK, int &ISTATE, int &IOPT, double *RWORK, int &LRW,
                int *IWORK, int &LIW,
                void (*JAC)(int&, double&, double*, int&, int&, double*, int&),
                int &JT, void (*G)(int&, double&, double*, int&, double*),
                int &NG, int *JROOT );
}

/**
177  * Run the system simulation while simultaneously feeding it input from the network simulation.
    */
int runSimulation( NeuralNetwork *n, double &fitness, stringstream *ss = 0 )
{
    n->reset(); // make sure the network is reset

```

```

    fitness = 0.0;
    int N = 0;

    double x[] = { ic[0], ic[1] }; // states
    double t = 0.0; // current time
187    double tout = 0.0; // next time
    int  istate = 1; // 1: first call, 2: subsequent calls, <0: errors
    double rwork[lrw]; // real work array
    int  iwork[liw]; // integer work array
    int  jroot[ng]; // boundary conditions

    /* set up output mechanism */
    deque<double> window; // rect window to calculate fitness
    deque<double> wtime;
    vector<double> outputs; // the value of each output neuron
197    U = 0;
    outputs.resize( noutput );
    fill( outputs.begin(), outputs.end(), 0.0 );

    /* Noise debugging */
#ifdef DEBUG_NOISE
    ofstream procoss;
    ofstream measoss;
    procoss.open("proc_noise.dat", ios::app);
    measoss.open("meas_noise.dat", ios::app);
207 #endif

    /* EWMA demodulation */
    double tk[noutput]; // last fire time
    double wk[noutput]; // average fire rate
    for ( int i = 0; i < noutput; i++ ) // initialization
    {
        tk[i] = 0.0;
        wk[i] = 0.0;
    }

217    /* Run Simulation */
    double next_queue_time;
    double next_time;
    while ( t < endtime )
    {
        tout += dtout; // next simulation time

```

```

        /* run the network simulation up to tout */
        while ( n->getNextTime() > 0 && n->getNextTime() < tout )
227 {
            double tmp = n->advanceSimulation( outputs );

            /* check if the queue has blown up */
            if ( n->getQueueSize() > q_tol )
            {
                fitness = SIMFAIL;
                return 1;
            }
            if ( n->getpulselistsize() > 10*q_tol )
237 {
                fitness = SIMFAIL;
                return 2;
            }

            /* demodulate output neurons by lowpass EWMA */
            for ( int i = 0; i < noutput; i++ )
            {
                if ( outputs[i] )
            {
247 double alpha = n->getOutputAlphaAt( i );
                if ( tk[i] > 0 )
                {
                    wk[i] = alpha/(tmp-tk[i]) + (1-alpha)*wk[i];
                    //          if ( outputs[i] < 0 )
                    //          wk[i] *= -1;
                }
                tk[i] = tmp;
            }
            }

257

            /* map demodulated outputs to dynamic system input(s) */
            U = outputs[0]*wk[0];

            /* reset output vector */
            fill( outputs.begin(), outputs.end(), 0.0 );
        }

        /* apply input noise */
        if ( pnoise )
267 {

```

```

        for ( int i = 0; i < 1; i++ )
        {
            proc_noise[i] = p_alpha*n->randf(-1,1) + (1-p_alpha)*proc_noise[i];
            proc_noise[i] *= 1.0;
#ifdef DEBUG_NOISE
            procoss << proc_noise[i] << " ";
#endif
        }
#ifdef DEBUG_NOISE
277     procoss << endl;
#endif

    U += proc_noise[0];
}

    /* run solver */
    dlsodar_( F,      neq,    x,      t,      tout,
            itol,  rtol,  abtol,
            itask, istate, iopt,  rwork, lrw,  iwork, liw,
287     JAC,   jt,      G,      ng,      jroot );

    if ( istate < 0 ) // an error occurred
    {
        //      cerr << "Warning: istate indicates LSODE failure, exiting simulation. " << endl;
        fitness = SIMFAIL;
        return 1;
    }

    else if ( istate == 2 && ss ) // save results and continue
    {
297     (*ss) << t << " " << x[0] << " " << x[1] << " " ;
        (*ss) << n->getInputAccAt( 0 ) << " " << n->getInputAccAt( 1 ) << " ";
        (*ss) << U << endl;
    }

    else if ( istate == 3 ) // boundary encountered
        istate = 2; // reset istate to continue

    /* check state limits (keep calculation from blowing up) */
    for ( int i = 0; i < neq; i++ )
    if ( state_tol[i] && ( x[i] > state_tol[i] || x[i] < -state_tol[i] ) )
307     {
        fitness = SIMFAIL;
        return 2;
    }
}

```

```

        /* apply measurement noise */
        if ( mnoise )
        {
            for ( int i = 0; i < neq; i++ )
            {
317         meas_noise[i] = m_alpha*n->randf(-1,1) + (1-m_alpha)*meas_noise[i];
#ifdef DEBUG_NOISE
            measos << meas_noise[i] << "␣";
#endif
        }
#ifdef DEBUG_NOISE
        measos << endl;
#endif
    }

327     /* State feedback */
    n->applyInputAt( 0, tout+dtout, x[0] + meas_noise[0] );
    n->applyInputAt( 1, tout+dtout, x[1] + meas_noise[1] );

    /* update FITNESS window */
    fitness += (x[0]*x[0]);
    N++;

    } // end while ( simulation )

337     fitness /= N;
    n->reset();
    return 0;
}

#endif

```

A.8 sim_pendulum_ctrl.h

```

/**
 * Author: Scott Hansen
 * File: sim_pendulum_ctrl.h
 * Date: Feb 16, 2012
 *
 */

8 #ifndef SIM_H

```

```

#define SIM_H

#include <string>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cstdlib>
#include <sstream>
#include <iostream>
18 #include <fstream>
#include <ostream>
#include <queue>
#include <set>
#include <deque>
using namespace std;

/* Debugging */
// #define DEBUG
// #define DEBUG_NOISE
28 // #define DEBUG_STATS
// #define DEBUG_MSE

/* System Parameters */
double M = 10.0;    // cart mass (Kg)
double m = 2;      // pendulum mass (Kg)
double L = 0.5;    // pendulum length (m)
double k = 0.1;    // friction coefficient
double g = 9.81;   // gravitational constant (m/s^2)
double I = m*L*L/3; // moment of inertia
38 double pi = 3.14159265358979;
#define SIMFAIL 100000000.0 // aka a really big number

/* I/O */
string netfile;

/* Neuralnetwork Properties */
int seed = time(NULL); // random seed
double density = 0.1; // connection density
double xsize = 1; // network x dimension size
48 double ysize = 1; // network y dimension size
double zsize = 1; // network z dimension size
int ninput = 1; // number of input neurons
int nhidden = 1; // (default) number of hidden neurons

```

```

int      noutput  = 1;          // number of output neurons
int      nidx     = 1;          // default network

/* EP Variables */
double    fitness_tol  = 1.0;
int        gen_max     = 100;
58 double    state_tol[] = { 0, 0, 0, 0 };
double    T            = 0.9; // SA temperature
double    fail_tol     = 0.3;
double    q_tol        = 1000;
double    w[]          = {0.5, 0.5}; //weight of fitness for each state and total fitness

/* DLSODAR Simulation Variables */
int        neq        = 4;      // number of first order diff eq's
int        ng         = 0;      // number of boundary condition equations
double     dtout       = 0.1;   // sampling time
68 int      itol        = 1;     // 1: scalar rep. for atol, 2: array rep. for each state
double     rtol        = 0.0;   // relative tolerance
double     abtol       = 0.0000001; // absolute tolerance
int        itask       = 1;     // 1: normal task specification
int        iopt        = 0;     // 0: no optional inputs, 1: optional inputs specified
int        lrw         = 32 + neq*(neq+9) + 3*ng; // length of real work array
int        liw         = 20 + neq; // length of integer work array
double     endtime     = 30;    // endtime
int        jt          = 2;     // 1: user supplied jac 2: internally generated jac
double     U           = 0.0;   // input
78 double   Unet        = 0.0;
double     err         = 0.0;   // nonlinear control signal
double     ic[]        = {pi + 0.1, 0.0, 0.0, 0.0}; // default initial conditions

/* Noise */
bool load_pnoise = false;
bool pnoise = true;
bool mnoise = false;
vector<double> proc_noise( ninput, 0 ); // one for each input
vector<double> meas_noise( neq, 0 );   // one for each state
88 double p_alpha = 0.5;
double m_alpha = 0.5;
double p_mag = 2.0;
int navg = 10; // for noise optimization, take and average of several runs

/* DLSODAR Functions */
void F( int &NEQ, double &T , double *X , double *XDOT )

```

```

{
    double xnoise[] = { xnoise[0] = X[0] + meas_noise[0],
                        xnoise[1] = X[1] + meas_noise[1],
98      xnoise[2] = X[2] + meas_noise[2],
                        xnoise[3] = X[3] + meas_noise[3] };

    /* Control law */
    double r = 0;
    double kp = 0.9;
    double E0 = 2*m*g*L; // desired energy
    double E = 1/2 * I*x[1]*x[1] + m*g*L*(1+cos(x[0]));
    //double V = ( E - E0 ) * ( E - E0 ) / 2;
    double e1 = 8.54088, e2 = 0.598677;

108
    // if ( T < 2.25 )
    if ( sqrt((E-E0)*(E-E0)) >= e1 && 1-cos(xnoise[0]) >= e2 )
    {
        r = kp * ( E - E0 ) * xnoise[1] * cos(xnoise[0]);
    }
    else // linear lqr stabilization for small signal
    {
        r = -( -258.9975*xnoise[0]
              -63.5652*xnoise[1]
118      -1.1050*xnoise[2]
              -5.1523*xnoise[3] );
    }

    //Unet = 0;
    err = r + proc_noise[0];
    U = Unet + err;

    // Khalil:
    double delta = (I+m*L*L)*(m+M) - (m*m*L*L)*cos(X[0])*cos(X[0]);
    double M11 = m+M;
    double M12 = -m*L*cos(X[0]);
128
    double M21 = M12;
    double M22 = I + m*L*L;
    double N1 = m*g*L*sin(X[0]);
    double N2 = U + m*L*X[1]*X[1]*sin(X[0]) - k*X[4];

    XDOT[0] = X[1];
    XDOT[1] = ( M11*N1 + M12*N2 ) / delta;
    XDOT[2] = X[3];
    XDOT[3] = ( M21*N1 + M22*N2 ) / delta;

```

```

138 }

void JAC( int &NEQ, double &T, double *X, int &ML, int &MU, double *PD, int &NROWPD )
{
}

void G( int &NEQ, double &T, double *X, int &NG, double *GOUT )
{
}

148 extern "C"
{
    void dlsodar_( void (*F)(int&, double&, double*, double*),
        int &NEQ, double *X, double &T, double &TOUT, int &ITOL, double &RTOL,
        double &ATOL, int &ITASK, int &ISTATE, int &IOPT, double *RWORK, int &LRW,
        int *IWORK, int &LIW,
        void (*JAC)(int&, double&, double*, int&, int&, double*, int&),
        int &JT, void (*G)(int&, double&, double*, int&, double*),
        int &NG, int *JROOT );
    void xsetf_( int &MFLAG );
158 }

/**
 * Run the system simulation while simultaneously feeding it input from the network simulation.
 */
int runSimulation( NeuralNetwork *n, double &fitness, stringstream *ss = 0)
{
    n->reset(); // make sure the network is reset

    fitness = 0.0;
168    int N = 0; // number of iterations
    double f[] = { 0.0, 0.0 }; // fitness

    double x[] = { ic[0], ic[1], ic[2], ic[3] }; // states
    double t = 0.0; // current time
    double tout = 0.0; // next time
    int  istate = 1; // 1: first call, 2: subsequent calls, <0: errors
    double rwork[lrw]; // real work array
    int  iwork[liw]; // integer work array
    int  jroot[ng]; // boundary conditions

178    /* set up output mechanism */
    vector<double> outputs; // the value of each output neuron

```

```

        outputs.resize( noutput );
        fill( outputs.begin(), outputs.end(), 0.0 );

        /* Noise debugging */
#ifdef DEBUG_NOISE
        ofstream procos;
        ofstream measos;
188     procos.open("proc_noise.dat" );
        measos.open("meas_noise.dat" );
#endif

#ifdef DEBUG_MSE
        ofstream mseos;
        mseos.open("mse.dat");
#endif

        /* EWMA demodulation */
198     double tk[noutput];
        double wk[noutput];
        double alpha = 1; // lowpass filter
        for ( int i = 0; i < noutput; i++ )
        {
            tk[i] = 0.0;
            wk[i] = 0.0;
        }

        // read dedicated proc noise
208     deque<double> dpnoise;
        if ( load_pnoise )
        {
            ifstream ifs;
            ifs.open("./proc_noise.dat");
            if ( ifs.is_open() )
            {
                double d;
                while ( ifs >> d )
                    dpnoise.push_back(d);
218     }
        }

        /* Run Simulation */
        double next_queue_time;
        double next_time;

```

```

while ( t < endtime )
{
    tout += dtout; // next simulation time

228     /* Generate measurement noise */
    if ( mnoise )
    {
        for ( int i = 0; i < neq; i++ )
        {
            meas_noise[i] = m_alpha*n->randf(-1,1) + (1-p_alpha)*meas_noise[i];
            meas_noise[i] *= 1.0;
#ifdef DEBUG_NOISE
                measos << meas_noise[i] << "␣";
#endif
238         }
#ifdef DEBUG_NOISE
            measos << endl;
#endif
        }

        /* Generate process noise */
        if ( pnoise )
        {
            for ( int i = 0; i < noutput; i++ )
248             {
                proc_noise[i] = p_alpha*n->randf(-1,1) + (1-p_alpha)*proc_noise[i];
                proc_noise[i] *= p_mag;
#ifdef DEBUG_NOISE
                    procoss << proc_noise[i] << "␣";
#endif
                }
#ifdef DEBUG_NOISE
                    procoss << endl;
#endif
258             }

            /* Run the network simulation up to tout */
            Unet = 0;
            while ( n->getNextTime() > 0 && n->getNextTime() < tout )
            {
                double tmp = n->advanceSimulation( outputs );

                /* check if the queue has blown up */

```

```

    if ( n->getQueueSize() > q_tol )
    {
        fitness = SIMFAIL;
        return 1;
    }
    if ( n->getpulselistsize() > 10*q_tol )
    {
        fitness = SIMFAIL;
        return 2;
    }

268
    for ( int i = 0; i < noutput; i++ ) // demod neurons
    {
        if ( outputs[i] )
        {
            double alpha = n->getOutputAlphaAt( i );
            if ( tk[i] > 0 )
            {
                wk[i] = alpha/(tmp-tk[i]) + (1-alpha)*wk[i];
                if ( outputs[i] < 0 )
                wk[i] *= -1;
288
            }
            tk[i] = tmp;
        }
    }

    Unet += outputs[0]*wk[0];
    fill( outputs.begin(), outputs.end(), 0.0 );
}

/* run solver */
298
int mflag = 0;
xsetf_( mflag ); // supress dlsodar output
dlsodar_( F,      neq,      x,      t,      tout,
itol,  rtol,  abtol,
itask, istate, iopt,  rwork, lrw,  iwork, liw,
JAC,   jt,      G,      ng,      jroot );

/* Normalize angle */
x[0] = fmod( x[0], 2*pi );

308
double deltax = x[0];
if ( deltax < 0 )

```

```

deltax *= -1;
    if ( 2*pi-deltax < deltax )
deltax = 2*pi - deltax;

    /* update fitness */
    if ( t > 0 )
{
    f[0] += (deltax*deltax); //(x[0]*x[0]);
318    f[1] += (x[2]*x[2]);
    N++;
}

#ifdef DEBUG_MSE
    mseos << deltax << " " << w[0]*f[0]/N + w[1]*f[1]/N << endl;
#endif
    if ( istate < 0 ) // an error occurred
    {
        fitness = SIMFAIL;
328    return 1;
    }

    else if ( istate == 2 && ss ) // save results and continue
    {
        (*ss) << t << " " << x[0] << " " << x[1] << " ";
        (*ss) << x[2] << " " << x[3] << " ";
        (*ss) << err << " " << Unet << " " << proc_noise[0] << endl;
    }

    else if ( istate == 3 ) // boundary encountered
    {
338    istate = 2; // reset istate to continue
    }

    /* check state limits (keep calculation from blowing up) */
    for ( int i = 0; i < neq; i++ )
if ( state_tol[i] && ( x[i] > state_tol[i] || x[i] < -state_tol[i] ) )
    {
        fitness = SIMFAIL;
        return 2;
    }

348

    /* Error feedforward */
    n->applyInputAt( 0, tout, err );
} // end while ( simulation )

```

```

#ifdef DEBUG_NOISE
    procos.close();
    measos.close();
#endif
#ifdef DEBUG_MSE
358     mseos.close();
#endif

    fitness = w[0]*f[0] + w[1]*f[1];
    fitness /= N;

    return 0;
}

void process_sim_arguments( int argc, char** argv )
368 {
    for ( int i = 1; i < argc; i++ )
    {
        if ( strcmp( argv[i], "-h" ) == 0 )
        {
            cerr << "just read the switches in the code, its not difficult..." << endl;
            exit(0);
        }
        else if ( strcmp( argv[i], "-seed" ) == 0 ) // network properties
            seed = atoi( argv[i+1] );
378     else if ( strcmp( argv[i], "-density" ) == 0 )
            density = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-xsize" ) == 0 )
            xsize = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-ysize" ) == 0 )
            ysize = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-zsize" ) == 0 )
            zsize = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-ninput" ) == 0 )
            ninput = atoi( argv[i+1] );
388     else if ( strcmp( argv[i], "-nhidden" ) == 0 )
            nhidden = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-noutput" ) == 0 )
            noutput = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-nidx" ) == 0 )
            nidx = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-fitness_tol" ) == 0 ) // EP variables
            fitness_tol = atof( argv[i+1] );

```

```

        else if ( strcmp( argv[i], "-statetol0" ) == 0 )
state_tol[0] = atof( argv[i+1] );
398         else if ( strcmp( argv[i], "-statetol1" ) == 0 )
state_tol[1] = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-statetol2" ) == 0 )
state_tol[2] = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-statetol3" ) == 0 )
state_tol[3] = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-genmax"      ) == 0 )
gen_max = atoi( argv[i+1] );
        else if ( strcmp( argv[i], "-dtout"       ) == 0 ) // DLSODAR variables
dtout = atof( argv[i] );
408         else if ( strcmp( argv[i], "-rtol"      ) == 0 )
rtol = atof( argv[i] );
        else if ( strcmp( argv[i], "-abtol"      ) == 0 )
abtol = atof( argv[i] );
        else if ( strcmp( argv[i], "-endtime"    ) == 0 )
{
    endtime = atof( argv[i+1] );
}
        else if ( strcmp( argv[i], "-ic0"        ) == 0 )
ic[0] = atof( argv[i+1] );
418         else if ( strcmp( argv[i], "-ic1"        ) == 0 )
ic[1] = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-ic2"        ) == 0 )
ic[2] = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-ic3"        ) == 0 )
ic[3] = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-pnoise"     ) == 0 ) // noise variables
pnoise = true;
        else if ( strcmp( argv[i], "-mnoise"     ) == 0 )
mnoise = true;
428         else if ( strcmp( argv[i], "-palpha"    ) == 0 )
p_alpha = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-pmag"      ) == 0 )
p_mag = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-malpha"    ) == 0 )
m_alpha = atof( argv[i+1] );
        else if ( strcmp( argv[i], "-d"         ) == 0 ) // other
netfile = argv[i+1];
    }
}
438

```

```
#endif
```

A.9 mainmpi2.cpp

```
1  /**
   *   File: main.cpp
   *   Author: Scott Hansen
   *   Date: Feb 14, 2012
   *
   */

#include <mpi.h>
#include "NeuralNetwork.h"
#include "NeuralNetworkFactory.h"
11 #include "Breeder.h"

/* Load Simulation Data and Functions */
#ifdef HARMONIC
#include "sim_harmonic.h"
#endif
#ifdef PENDULUM
#include "sim_pendulum.h"
#endif

21 /* OpenMPI Job Tags */
#define DIETAG      1 // instruct the slave to exit
#define WORKTAG     2 // business as usual

/* OpenMPI Variables */
int rank = 0; // process rank
MPI_Status status;

void master( ostream &out = cout ); // the bourgeoisie
void slave( ostream &out = cout ); // the proletariat

31 /* Genetic unit */
struct Chromosome
{
    Chromosome() : c(), f(), s() {}
    Chromosome( string const& c_in, double f_in, double s_in ) : c(c_in), f(f_in), s(s_in) {}
    bool operator< ( Chromosome const& rhs ) const { return f < rhs.f; }
    string c; // the chromosome
    double f; // fitness
}
```

```

        double s; // source
41  };

    /*
    * Main function.
    */
    int main ( int argc, char** argv )
    {
        process_sim_arguments( argc, argv );
        srand(seed);

51  /* run a single file specified by the -d flag */
        if ( !netfile.empty() )
        {
            cout << "reading_" << netfile << endl;
            NeuralNetwork *network;
            NeuralNetworkFactory nnf;
            stringstream ss;
            ofstream os;
            double fitness;
            double flag;
61         ifstream ifs;

            ifs.open( netfile.c_str() );
            if ( ifs.is_open() )
            {
                network = new NeuralNetwork( ifs );
                network->print();
                ifs.close();
            }
            else
71         {
                cerr << "Error: bad_file_name, loading the default network." << endl;
                network = nnf.createNetwork( nidx, ninput, nhidden, noutput, density );

                os.open( "netrun.net" );
                if ( os.is_open() )
                    network->write( os );
                os.close();
            }

81         flag = runSimulation( network, fitness, &ss );
            cout << "Ran_file_" << "flag_" << flag << "fitness_" << fitness << endl;

```

```

        string file = "netrun.dat";
        os.open( file.c_str() );
        if ( os.is_open() )
        os << ss.str();
        os.close();
        cout << "File saved to " << file << endl;

        delete network;
91     return 0;
    }

    /* Run parallel program */
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    if ( rank == 0 )
    {
    #ifndef DEBUG
101         ofstream ofs;
            ofs.open( "./output/out.log" );
            master( ofs );
            ofs.close();
    #else
            master();
    #endif
    }
    else
    {
        slave();
111
        MPI_Finalize();
        // return 0;
    }

    /**
        * Manage population and structural changes.
    */

    void master( ostream &out )
    {
121     out << "checking in from master " << rank << ", ";
        int nproc; // there are nproc-1 slaves
        MPI_Comm_size( MPI_COMM_WORLD, &nproc );

        multiset<Chromosome> population;

```

```

NeuralNetworkFactory nnf;
double best_fitness = SIMFAIL;
int count = 1;
int generation;
int psize = nproc-1; // create population equal to the number of slaves available
131 out << psize << "▯slaves▯available" << endl;

#ifdef DEBUG_STATS
    vector<int> num_neurons(50);
    vector<int> num_synaps(50);
    vector<int> num_synaps_in_per_neuron(50);
    vector<int> num_synaps_out_per_neuron(50);
    clock_t t1, t2;
    t1 = clock();
#endif

141 /* Initialize population */
for ( int i = 0; i < psize; i++ )
{
    ifstream ifs;
    stringstream ss;
    NeuralNetwork *n;
    double m;

    // save population member
151 // ifs.open("./proc0.net");
    // n = new NeuralNetwork( ifs );
    // ifs.close();
    n = nnf.createNetwork( nidx, ninput, nhidden, noutput, density );
    n->write( ss );
    runSimulation( n, m );
    Chromosome c( ss.str(), m, 0 );
    population.insert( c );
    delete n;
}

161 out << "population▯initialized▯to▯" << population.size() << "▯chromosomes" << endl;

/* EP/GA */
generation = 0;
count;
while( best_fitness > fitness_tol )
{
    out << "->generation▯" << generation << endl;

```

```

        /* Send out jobs */
171     count = 1;
        for ( set<Chromosome>::iterator itr = population.begin(); itr != population.end(); itr++ )
        {
            stringstream ss( itr->c );
            NeuralNetwork *n = new NeuralNetwork( ss );

            // mutate population member
            Breeder b(n);
            int r = b.mutate_structure();
            while ( r < 0 )
181         r = b.mutate_structure();

#ifdef DEBUG_STATS
            num_neurons[ n->getNumberOfNeurons() ]++;
            num_synaps[ n->getNumberOfConnections() ]++;
            n->countSynapPerNeuron( num_synaps_in_per_neuron, num_synaps_out_per_neuron );
#endif

            // send to slave
            stringstream sr;
191     n->write( sr );
            MPI_Send( (void*)sr.str().c_str(),
                      sr.str().length(),
                      MPI_CHAR,
                      count,
                      WORKTAG,
                      MPI_COMM_WORLD );

            count++;
        }

201
        /* Recieve results */
        count = 0;
        while ( count < psize )
        {
            double fitness;
            int source;
            char *buffer;
            int buffer_len;
            MPI_Recv( &fitness,
211         1,

```

```

        MPI_DOUBLE,
        MPI_ANY_SOURCE,
        WORKTAG,
        MPI_COMM_WORLD,
        &status );
source = status.MPI_SOURCE;

MPI_Probe( source, MPI_ANY_TAG, MPI_COMM_WORLD, &status );

221 MPI_Get_count( &status, MPI_CHAR, &buffer_len );
    buffer = new char[ buffer_len ];
    MPI_Recv( buffer,
              buffer_len,
              MPI_CHAR,
              source,
              WORKTAG,
              MPI_COMM_WORLD,
              &status );

231 string str( buffer );
    Chromosome c( str, fitness, source );
    population.insert( c );

#ifdef DEBUG
    out << "recieved_fitness=" << fitness << "from_slave" << source << "\n";
    out << "(pop" << population.size() << ") \n" << endl;
#endif
    // clean up
    delete[] buffer;
241 count++;
}

#ifdef DEBUG
    out << "finished_recieving_from_slaves" << endl;
#endif

    /* Drop poor performers */
    count = 0;
    for ( set<Chromosome>::iterator itr = population.begin(); itr != population.end(); itr++ )
    {
        if ( count == psize )
251     {
            population.erase( itr, population.end() );
            break;
        }
    }

```

```

        count++;
    }

    /* Save best results if an improvement is found */
    if ( population.begin()->f < best_fitness )
    {
261      out << "improvement_ found, fitness_=" << population.begin()->f << " ";
      out << "from_ " << population.begin()->s << "!" << endl;
      stringstream ss;
      ss << "./output/best.net"; // _ << generation << ".net";
      ofstream os;
      os.open( ss.str().c_str() );
      if ( os.is_open() )
          os << population.begin()->c;
      os.close();
      best_fitness = population.begin()->f;
271      out << "saved_ improvement_ " << best_fitness << "!" << endl;
    }

    generation++;
} // end while

#ifdef DEBUG_STATS
    t2 = clock();
    ofstream ofs;
    ofs.open("./output/stats.dat");
281    ofs << fitness_tol << " ";
    ofs << population.size() << " ";
    ofs << ((float)t2-(float)t1)/CLOCKS_PER_SEC << " ";
    ofs << generation-1 << endl;
    for ( int i = 0; i < num_neurons.size(); i++ )
    {
        ofs << i << " " << num_neurons[i] << " " << num_synaps[i] << " ";
        ofs << num_synaps_in_per_neuron[i] << " ";
        ofs << num_synaps_out_per_neuron[i] << endl;
    }
291    ofs.close();
#endif

/* Send out dietags to slaves */
for ( int i = 1; i < nproc; i++ )
    MPI_Send( 0,
              0,

```

```

        MPI_INT,
        i,
        DIETAG,
301      MPI_COMM_WORLD );

    return;
}

/**
 * Manages parameter changes.
 */
void slave( ostream &out )
{
311   while ( 1 )
    {
        /* Recieve network from master */
        MPI_Probe( 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status );

        if ( status.MPI_TAG == DIETAG )
            break;

        char *buffer = 0;
        int buffer_len;
321      MPI_Get_count( &status, MPI_CHAR, &buffer_len );
        buffer = new char[ buffer_len ];
        MPI_Recv( buffer,
            buffer_len,
            MPI_CHAR,
            0,
            WORKTAG,
            MPI_COMM_WORLD,
            &status );

331      /* Parse and Evaluate */
        NeuralNetwork *n;
        stringstream best;
        double best_fitness;
        best.str( buffer );
        n = new NeuralNetwork( best ); // initial slave network
        runSimulation( n, best_fitness );

        /* SA to search for a better network */
        double fitness = 0.0;

```

```

341     double K = 1.0; // SA temperature
        int flag;
        int simerr = 0;
        for ( int i = 0; i < gen_max; i++ )
        {

            // mutate
            Breeder b( n );
            int result = -1;
            while ( result < 0 )
351         result = b.mutate_parameter();

            // run simulation
            flag = runSimulation( n, fitness );

#ifdef DEBUG
            out << "slave_" << rank << "_";
            out << "itr_" << i << "_";
            out << "fitness_" << fitness << "_";
            out << "flag_" << flag << "_";
361     #endif

            // check for errors
            if ( flag )
            {
                simerr++;
                if ( simerr > fail_tol*gen_max ) // abort!
                    break;
            }

371     // evaluate performance
            K *= T;
            if ( fitness < best_fitness )
            {
                stringstream sr;
                n->write( sr );
                best_fitness = fitness;
                best.str( sr.str() );

#ifdef DEBUG
381         out << "improvement!" << endl;
#endif
            }
}

```

```

        else if ( (double)rand()/RAND_MAX < K ) // anneal
        {
#ifdef DEBUG
            out << "annealing..."<< endl;
#endif
        }
        else
391     {
            stringstream sr( best.str() );
            n->read( sr );
#ifdef DEBUG
            out << "replacing_ with_" << best_fitness << endl;
#endif
        }

    } // end for

401     /* Send best to master */
#ifdef DEBUG
        out << "sending_" << best_fitness << endl;
#endif

        MPI_Send( &best_fitness,
            1,
            MPI_DOUBLE,
            0,
            WORKTAG,
411     MPI_COMM_WORLD );
        MPI_Send( (void*)best.str().c_str(),
            best.str().length(),
            MPI_CHAR,
            0,
            WORKTAG,
            MPI_COMM_WORLD );

        /* Clean up */
        delete[] buffer;
421     delete n;

    } // end while ( 1 )

    return;
}

```

A.10 pendulum_ctrl.cpp

```
/**
 *   File: pendulum_ctrl.cpp
 *   Author: Scott Hansen
4  *   Date: Feb 14, 2012
 *
 */

#include <mpi.h>
#include "NeuralNetwork.h"
#include "NeuralNetworkFactory.h"
#include "Breeder.h"

/* Load Simulation Data and Functions */
14 #include "sim_pendulum_ctrl.h"

/* OpenMPI Job Tags */
#define DIETAG      1 // instruct the slave to exit
#define WORKTAG     2 // business as usual

/* OpenMPI Variables */
int rank = 0; // process rank
MPI_Status status;

24 void master( ostream &out = cout ); // the bourgeoisie
void slave( ostream &out = cout ); // the proletariat

/* Genetic unit */
struct Chromosome
{
    Chromosome() : c(), f(), s() {}
    Chromosome( string const& c_in, double f_in, double s_in ) : c(c_in), f(f_in), s(s_in) {}
    bool operator< ( Chromosome const& rhs ) const { return f < rhs.f; }
    string c; // the chromosome
34    double f; // fitness
    double s; // source
};

/*
 * Main function.
 */
int main ( int argc, char** argv )
```

```

{
    process_sim_arguments( argc, argv );
44    srand(seed);

    /* run a single file specified by the -d flag */
    if ( !netfile.empty() )
    {
        cout << "reading_" << netfile << endl;
        NeuralNetwork *network;
        NeuralNetworkFactory nnf;
        stringstream ss;
        ofstream os;
54        double fitness;
        double flag;
        ifstream ifs;

        ifs.open( netfile.c_str() );
        if ( ifs.is_open() )
        {
            network = new NeuralNetwork( ifs );
            network->print();
            ifs.close();
64        }
        else
        {
            cerr << "Error: bad_file_name, loading_network_" << nid << endl;
            network = nnf.createNetwork( nid, nin, nhid, nout, density );

            os.open( "netrun.net" );
            if ( os.is_open() )
                network->write( os );
            os.close();
74        }

        flag = runSimulation( network, fitness, &ss );
        cout << "Ran_file_" << "flag_" << flag << "fitness_" << fitness << endl;
        string file = "netrun.dat";
        os.open( file.c_str() );
        if ( os.is_open() )
            os << ss.str();
        os.close();
        cout << "File_saved_to_" << file << endl;
84

```

```

        delete network;
        return 0;
    }

    // process_sim_arguments( argc, argv );
    // NeuralNetwork *n = 0;
    // double mse;
    // stringstream ss;
    // int flag;

94    // flag = runSimulation( n, mse, &ss );
    // cout << "simulation quit with flag " << flag << endl;

    // ofstream ofs;
    // ofs.open("./output/output.dat");
    // ofs << ss.str();
    // ofs.close();

    /* Run parallel program */
104    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if ( rank == 0 )
    {
        #ifndef DEBUG
            ofstream ofs;
            ofs.open( "./output2/out.log" );
            master( ofs );
            ofs.close();
        #else
114            master();
        #endif
    }
    else
    {
        slave();

        MPI_Finalize();
        // return 0;
    }

124 /**
    * Manage population and structural changes.
    */
    void master( ostream &out )

```

```

{
    out << "checking_in_from_master_" << rank << ",_";
    int nproc; // there are nproc-1 slaves
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );

    multiset<Chromosome> population;
134    NeuralNetworkFactory nnf;
    double best_fitness = SIMFAIL;
    int count = 1;
    int generation;
    int psize = nproc-1; // create population equal to the number of slaves available
    out << psize << "_slaves_available" << endl;

#ifdef DEBUG_STATS
    vector<int> num_neurons(50);
    vector<int> num_synaps(50);
144    clock_t t1, t2;
    t1 = clock();
#endif

    /* Initialize population */
    for ( int i = 0; i < psize; i++ )
    {
        stringstream ss;
        NeuralNetwork *n;
        double fitness;

154        // save population member
        ifstream ifs;
        ifs.open("./output2/iamapendulumandsocanyou/pend_1_5.net");
        if ( ifs.is_open() )
        {
            n = new NeuralNetwork( ifs );
            ifs.close();
            n->write( ss );
        }
164        else
        {
            cerr << "Error:_failed_to_load_file..." << endl;
            exit( 0 );
        }

        //      n = nnf.createNetwork( nidx, ninput, nhidden, noutput, density );

```

```

        // run simulation
        fitness = 0;
174     for ( int j = 0; j < navg; j++ )
    {
        double tmp = 0.0;
        runSimulation( n, tmp );
        fitness += tmp;
    }
        fitness /= navg;

        Chromosome c( ss.str(), fitness, 0 );
        population.insert( c );
184     delete n;
    }

    /* EP/GA */
    generation = 0;
    count;
    while( best_fitness > fitness_tol )
    {
        out << "->generation_□" << generation << endl;

194     /* Send out jobs */
        count = 1;
        for ( set<Chromosome>::iterator itr = population.begin(); itr != population.end(); itr++ )
    {
        stringstream ss( itr->c );
        NeuralNetwork *n = new NeuralNetwork( ss );

        // mutate population member
        Breeder b(n);
        int r = b.mutate_structure();
204     while ( r < 0 )
        {
            r = b.mutate_structure();
        }

#ifdef DEBUG_STATS
        num_neurons[ n->getNumberOfNeurons() ]++;
        num_synaps[ n->getNumberOfConnections() ]++;
#endif

        // send to slave
        stringstream sr;

```

```

214     n->write( sr );
        MPI_Send( (void*)sr.str().c_str(),
                  sr.str().length(),
                  MPI_CHAR,
                  count,
                  WORKTAG,
                  MPI_COMM_WORLD );

        count++;
    }

224     /* Recieve results */
        count = 0;
        while ( count < psize )
    {
        double fitness;
        int source;
        char *buffer;
        int buffer_len;
        MPI_Recv( &fitness,
234             1,
                MPI_DOUBLE,
                MPI_ANY_SOURCE,
                WORKTAG,
                MPI_COMM_WORLD,
                &status );
        source = status.MPI_SOURCE;

        MPI_Probe( source, MPI_ANY_TAG, MPI_COMM_WORLD, &status );

244     MPI_Get_count( &status, MPI_CHAR, &buffer_len );
        buffer = new char[ buffer_len ];
        MPI_Recv( buffer,
                  buffer_len,
                  MPI_CHAR,
                  source,
                  WORKTAG,
                  MPI_COMM_WORLD,
                  &status );

254     string str( buffer );
        Chromosome c( str, fitness, source );
        population.insert( c );

```

```

#ifdef DEBUG
out << "recieved_fitness=" << fitness << "from_slave" << source << " ";
out << "(pop" << population.size() << ")" << endl;
#endif

    // clean up
    delete[] buffer;
264    count++;
}

#ifdef DEBUG
out << "finished_recieving_from_slaves" << endl;
#endif

    /* Drop poor performers */
    count = 0;
    for ( set<Chromosome>::iterator itr = population.begin(); itr != population.end(); itr++ )
    {
274    if ( count == psize )
        {
            population.erase( itr, population.end() );
            break;
        }
        count++;
    }

    /* Save best results if an improvement is found */
    if ( population.begin()->f < best_fitness )
284    {
        out << "improvement_found_fitness=" << population.begin()->f << " ";
        out << "from" << population.begin()->s << "!" << endl;
        stringstream ss;
        ss << "./output2/best.net"; //_ << generation << ".net";
        ofstream os;
        os.open( ss.str().c_str() );
        if ( os.is_open() )
            os << population.begin()->c;
        os.close();
294    best_fitness = population.begin()->f;
        out << "saved_improvement" << best_fitness << "!" << endl;
    }

    generation++;
} // end while

```

```

#ifdef DEBUG_STATS
    t2 = clock();
    ofstream ofs;
304    ofs.open("./output2/stats.dat");
    ofs << fitness_tol << "\n";
    ofs << population.size() << "\n";
    ofs << ((float)t2-(float)t1)/CLOCKS_PER_SEC << "\n";
    ofs << generation-1 << endl;
    for ( int i = 0; i < num_neurons.size(); i++ )
    {
        ofs << i << "\n" << num_neurons[i] << "\n" << num_synaps[i] << endl;
    }
    ofs.close();
314 #endif

    /* Send out dietags to slaves */
    for ( int i = 1; i < nproc; i++ )
        MPI_Send( 0,
                  0,
                  MPI_INT,
                  i,
                  DIETAG,
                  MPI_COMM_WORLD );

324    return;
}

/**
 * Manages parameter changes.
 */
void slave( ostream &out )
{
    while ( 1 )
334    {
        /* Recieve network from master */
        MPI_Probe( 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status );

        if ( status.MPI_TAG == DIETAG )
            break;

        char *buffer = 0;
        int buffer_len;

```

```

        MPI_Get_count( &status, MPI_CHAR, &buffer_len );
344     buffer = new char[ buffer_len ];
        MPI_Recv( buffer,
buffer_len,
MPI_CHAR,
0,
WORKTAG,
MPI_COMM_WORLD,
&status );

        /* Parse and Evaluate */
354     NeuralNetwork *n;
        stringstream best;
        double best_fitness = 0.0;
        best.str( buffer );
        n = new NeuralNetwork( best ); // initial slave network

        for ( int i = 0; i < navg; i++ )
    {
        double tmp = 0.0;
        runSimulation( n, tmp );
364     best_fitness += tmp;
    }

        best_fitness /= navg;

        /* SA to search for a better network */
        double fitness = 0.0;
        double K = 1.0; // SA temperature
        int flag;
        int simerr = 0;
374     for ( int i = 0; i < gen_max; i++ )
    {
        // mutate
        Breeder b( n );
        int result = -1;
        while ( result < 0 )
            result = b.mutate_parameter();

        // run simulation
        fitness = 0.0;
384     for ( int j = 0; j < navg; j++ )
        {

```

```

        double tmp = 0.0;
        flag = runSimulation( n, tmp );
        fitness += tmp;

        if ( flag )
            simerr++;
    }
    fitness /= navg;

394
#ifdef DEBUG
    out << "slave_" << rank << "_";
    out << "itr_" << i << "_";
    out << "fitness_" << fitness << "_";
    out << "flag_" << flag << "_";
#endif

    // check for errors
    if ( simerr > fail_tol*gen_max*navg ) // abort!
404        break;

    // evaluate performance
    K *= T;
    if ( fitness < best_fitness )
    {

        stringstream sr;
        n->write( sr );
        best_fitness = fitness;
414        best.str( sr.str() );

#ifdef DEBUG
        out << "improvement!" << endl;
#endif
    }

    else if ( (double)rand()/RAND_MAX < K ) // anneal
    {
#ifdef DEBUG
        out << "annealing..." << endl;
424 #endif
    }
    else
    {
        stringstream sr( best.str() );

```

```

        n->read( sr );
#ifdef DEBUG
        out << "replacing_ with_" << best_fitness << endl;
#endif
    }
434 } // end for

        /* Send best to master */
#ifdef DEBUG
        out << "sending_" << best_fitness << endl;
#endif
        MPI_Send( &best_fitness,
            1,
            MPI_DOUBLE,
444         WORKTAG,
            MPI_COMM_WORLD );
        MPI_Send( (void*)best.str().c_str(),
            best.str().length(),
            MPI_CHAR,
            0,
            WORKTAG,
            MPI_COMM_WORLD );

        /* Clean up */
454 delete[] buffer;
        delete n;
    } // end while ( 1 )

    return;
}

```

A.11 makefile

```

CC= g++
MPICXX= mpic++
CFLAGS= -g
TARGETS= harmonic pendulum pendulum_ctrl
FLIB= /usr/lib/gcc/x86_64-linux-gnu/4.3.5/

all: $(TARGETS)

pendulum_ctrl: pendulum_ctrl.o NeuralNetworkFactory.o NeuralNetwork.o Breeder.o

```

```

10    $(MPICXX) $(CFLAGS) -o pendulum_ctrl pendulum_ctrl.o NeuralNetworkFactory.o NeuralNetwork.o Breeder.o

pendulum_ctrl.o: pendulum_ctrl.cpp sim_pendulum_ctrl.h
    $(MPICXX) -c pendulum_ctrl.cpp $(CFLAGS) -o pendulum_ctrl.o

harmonic: mainmpi2_harmonic.o NeuralNetworkFactory.o NeuralNetwork.o Breeder.o
    $(MPICXX) $(CFLAGS) -o harmonic mainmpi2_harmonic.o NeuralNetworkFactory.o NeuralNetwork.o Breeder.o

mainmpi2_harmonic.o: mainmpi2.cpp sim_harmonic.h
    $(MPICXX) -c mainmpi2.cpp $(CFLAGS) -D HARMONIC -o mainmpi2_harmonic.o
20

pendulum: mainmpi2_pendulum.o NeuralNetworkFactory.o NeuralNetwork.o Breeder.o
    $(MPICXX) $(CFLAGS) -o pendulum mainmpi2_pendulum.o NeuralNetworkFactory.o NeuralNetwork.o Breeder.o

mainmpi2_pendulum.o: mainmpi2.cpp sim_pendulum.h
    $(MPICXX) -c mainmpi2.cpp $(CFLAGS) -D PENDULUM -o mainmpi2_pendulum.o

.cpp.o:
    $(CC) -c $(CFLAGS) $< -o $@

30 .PHONY: clean

clean:
    rm -f *.o *~ ./output/* $(TARGETS)

tar:
    echo "backing up..."
    tar cfz backup-`date +%Y%m%d`.tar.gz *.cpp *.h makefile visualizer/*.pro visualizer/*.cpp visuali
    mv *.tar.gz ../archive/

```

Vita

Scott Frederick Hansen was born in Knoxville, Tennessee. After completing his studies at Farragut High School in 2006 he enrolled at the University of Tennessee, Knoxville. He received the degree of Bachelor of Science in Electrical Engineering from the University of Tennessee in 2010. During his undergraduate career he worked as an undergraduate research assistant at the Laboratory for Information Technologies. He continued his education at the University of Tennessee as a graduate student, working as a graduate research assistant and lecturer in electrical engineering.