SEQUENTIAL VERSUS EVENT-DRIVEN CONTROL NETWORKS: THE EFFECTS OF ORDER OF OPERATION WITH EXAMPLES FROM BIOLOGY

by

Kory A. Hardiman

Submitted in partial fulfilment of the requirements
for the degree of Master of Science

at

Dalhousie University
Halifax, Nova Scotia
July 2014

This thesis is dedicated to my parents, Richard and Michele, for their constant support and encouragement throughout all of my life.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This thesis provides a comparison between sequential and event-driven network simulation outcomes for biological control networks. A comparison is made between the effects of order of operation on these networks based on the two models. The sequential network model provides a serial processing approach using the Java programming language. The event-driven network model is executed using software manufactured by Trihedral Engineering Limited, called Visual Tag System (VTS). The thesis first describes any research completed in this area to date. Next, the thesis provides a detailed explanation of the mathematical model of the two different networks (sequential and event-driven) and the simulation strategies employed to obtain results. This is followed by a detailed analysis of the simulation outcomes. In addition, the models for sequential and event-driven both consider network plasticity and neural plasticity. Phenomena such as an ischemic event (heart attack) and the idea of twins' neural responses are examined.

LIST OF ABBREVIATIONS AND SYMBOLS USED

N.S.	Nova Scotia
j	Layer index
k	Neuron index
s	Neuron activity
N	Number of neurons per layer
n	Counting index
t	Time
$t^{(n)}$	Time at *(n)*
$\overline{F}^{(n)}$	Average neuron activity (function of time)
$S^{(n)}_{j,k}$	Neuron k activity in layer j
$\Delta M^{(n)}(t)$	Incremental move at time *(n)* (function of time)
β	Positive gain
α	Reference level
$M^{(n)}(t)$	Net move (function of time)
i	Counting index
τ_H	Process-time constant (for heart rate model)
$H^{(n)}(t)$	Heart rate as a function of time
δ	Neighboring neuron activity effect
H	Heart rate
D	Blood demand
$H^{(n)}_{j,k}$	Heart rate at time *(n)* for neuron k in layer j
$D^{(n)}_{j,k}$	Blood demand at time *(n)* for neuron k in layer j
L	Time shift constant
τ_L	Process-time constant (for sensory feedback updates)
$\overline{d}^{(n)}$	Average blood demand at time *(n)*
D_{max}	Maximum blood demand
$P^{(n)}_{j,k}$	Neighbor weights for neuron k in layer j
$U^{(n)}_{j,k}$	Number of neuron updates at time *(n)* for neuron k in layer j
$\overline{U}_{j,k}$	Average number of neuron updates for neuron k in layer j
$P^{(n)}_{j}$	Average layer importance at time *(n)* for layer j

ACKNOWLEDGEMENTS

This thesis would never have been started or completed without the positive inspiration from my parents. Their constant drive for me to better myself and persist to reach a final goal is truly admirable. They have always been true supporters of my work, no matter what I was working on. I will forever be in their debt for their encouragement and ability to create an environment that I could excel in.

Secondly, I would like to thank my fiancé and future bride, Meghan. Her love and constant words of encouragement gave me the energy to finish this thesis. Many nights were spent with us apart while this thesis and its research were being completed. And for that, she loves me nonetheless.

Lastly, I need to thank Dr. Guy Kember. We've been working together on all sorts of things for the last eight or so years. It has been quite a journey, from heart controllers to teaching first year university students how to take a derivative. I will always treasure the time we spent together, for it has made me a better problem solver. Additional thanks to the NSERC Discovery Grant Program that funded this research effort.

CHAPTER 1 INTRODUCTION

Mathematical models are typically linked with measured data by choosing parameters that give some form of *best-fit* between the measured data and the output from a mathematical model. In most simulations of mathematical models, the dependence of the numerical solution of a mathematical model on the orders of operation is assumed to be very small. This is mainly ensured by using a very small tolerance for error in the execution of solving such a model; for example, in space and time that provides convergent results. However, we shall observe that orders of operation have the capacity in the numerical solution of a nonlinear model to alter both the dynamics and the selection of steady-states and this is independent of the grid; for example, in space and time. In this thesis, two simulation types are considered: (i) sequential simulation where the orders of operation are not changing between simulations and (ii) event-driven simulation where the orders of operation are changing between simulations.

The introduction is divided into three subsections. First, a qualitative description of the mathematical model used in this thesis is provided. Next, a subsection is provided that explains the motivation for completing the work presented in this thesis. Finally, we examine the existing literature for any relevant ties to the work presented here.

1.1 QUALITATIVE MATHEMATICAL MODEL

The cardiac network model used in the simulations is outlined in Chapter 2 and is identical to that used in Kember *et al* [1]. In this subsection, we describe the mathematical model in qualitative terms. The controlled variable is the heart, which is assumed to follow a first-order, linear, time-invariant process. A qualitative representation of the model is provided in Figure 1.1. In the model, the heart rate is dimensionless and scaled to run between zero and one, *i.e.* a value of zero represents the typical low heart rate of an individual (45 beats per minutes) and a value of one represents the typical maximum heart rate (165 beats per minute) an individual can experience. The minimum neuron activity is zero and the maximum activity of any neuron is scaled to one. Every neuron saturates in nature [1], and this is given a value of

unity. Heart rate is fed back from the controlled variable to heart rate neurons and the external input is fed back to blood demand neurons. Each neuron also has neighboring neurons whose activities are dependent on the difference in their neighbors' activity and their own. In our simulations we considered two types of neural networks based on type of neuron and connectedness of neurons.

There are two types of neuron networks: (i) constant neural type and (ii) variable neural type. The constant neuron type is when every neuron in the simulation is of both heart rate and blood demand neurons. In contrast, the variable neural type network utilizes a mix of heart rate and blood demand neurons: blood demand neurons are concentrated at central command layer (see Figure 1.1) and heart rate neurons are concentrated at the cardiac layer, lastly, there is an equal mix between heart rate and blood demand neurons in the intra-thoracic layer. Connectedness is divided into two limiting cases: (i) sparse connectedness, where the number of neural activities applied to a neighbor is small compared to the number of neurons in the layer and (ii) maximally connected networks, where the number of neural activities applied to a neighbor approaches the number of neurons in a layer.

Plasticity is also considered and implemented by varying synaptic weights between neighboring neurons (termed network plasticity) and/or varying the heart importance and the blood demand importance (termed neural plasticity).

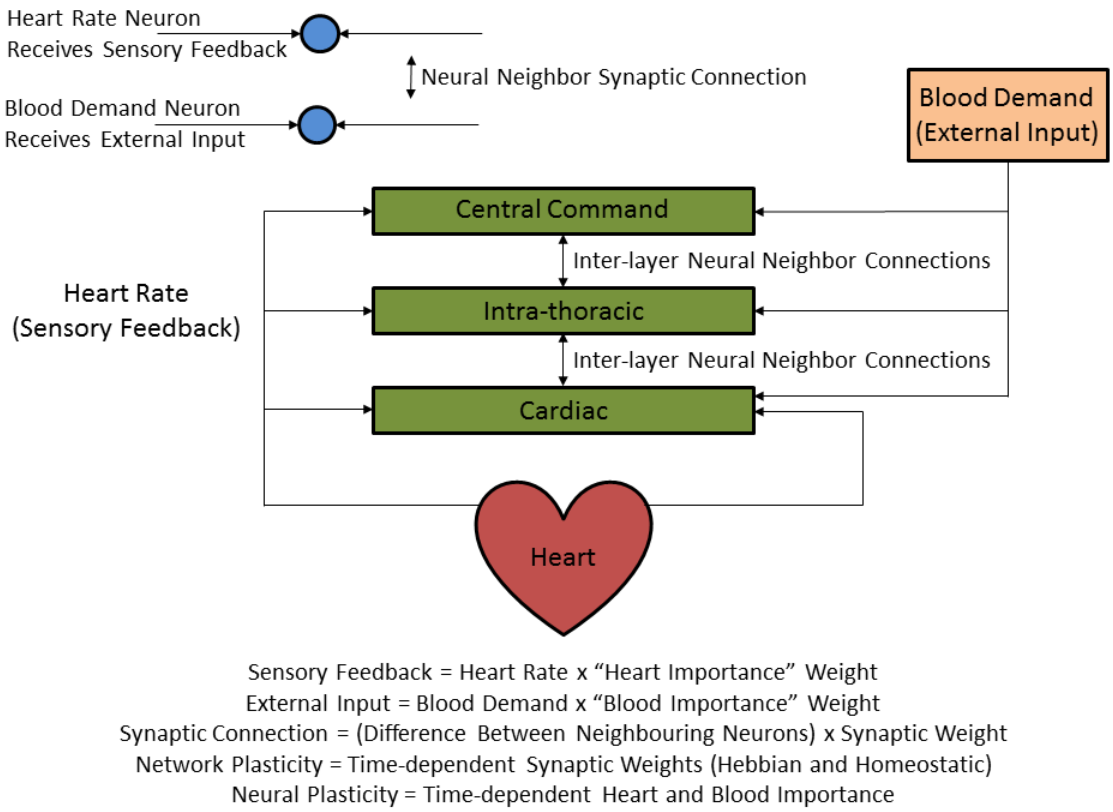

Figure 1.1: Schematic of the three-layered neural-networked cardiac control hierarchy. The heart rate is the controlled variable and is modified to meet the blood demand (external input). The control loop is closed by feeding back heart rate into the neural network. The neural network is of two kinds in this thesis: (i) constant neuron type where external input (blood demand) and heart rate sensory feedback are both supplied to every neuron in the neural hierarchy and (ii) variable neuron type where external input and heart rate sensory feedback are mixed in the network. A network is considered to be sparsely connected if there are few neighbor synaptic connections compared to the number of neurons in a layer. Maximally connected networks are where the number of interactions from neighboring neuron weights approaches that of the total number of neurons in a layer. Plasticity is implemented by making the neural and/or network plasticity time-dependent. Sequential versus event-driven simulations refer only to the basis by which we decide to update the status of a particular neuron in the network.

1.2 Thesis Motivation

The motivation of this thesis is to describe the effect of orders of operation on solution of nonlinear mathematical models. Orders of operation in this thesis refer specifically to the order in which the neural activities of neurons in the network are updated. The key point about the model described above is that it is strongly nonlinear primarily because the neuron activity is constrained to run between a minimum of zero and saturated value of one. This means that there are effectively an infinite number of steady-state solutions and

highly variable dynamics that are possible between steady-states. The selection of a steady-state and the way in which the solution tends to that steady-state is highly dependent on the orders of operation.

In the thesis we consider two methods of simulation: (i) sequential simulation and (ii) event-driven simulation. In our sequential simulations, we randomly chose neuron that is being updated with replacement. In contrast, in the event-driven simulations the decision to an update the activity of a neuron is based on a set of two different event-based criteria. The first is deemed the *watch interrupt model* where a neuron's activity is updated if any of its neuron activities change. The watch interrupt model is equivalent to looking for any change without any priority in the network. The other model is deemed the *large change interrupt model* where neuron's activities are updated if its change is beyond a predetermined threshold. This complements the watch interrupt model. Each of these event-driven models is used for network simulation outcomes while varying both neuron type (constant versus variable) networks and connected network type (sparse versus maximally-connected).

In clinical practice, heart rate (among other things) and its variability are used as markers to predict and measure the onset and progression of cardiac pathology. Clinical studies abound (Kember *et al* [1]) attempt to link population statistics with heart rate and its variability. These studies have found only inconsistent and inconclusive results. The results of this thesis go to the core of why it may not be possible to find any consistent link between heart rate variability and cardiac pathology at the level of a population or an individual. Specifically, we show that the clinical outcome for a pair of identical twins subjected to the same cardiac pathology may be completely different due to changes in the dynamics of internal network processing. In our model we consider *twin networks* that share the same initial properties and initial neural activities.

Partway through each simulation, we subject the network to a cardiac ischemia (heart attack) that is modelled as an increase in the blood demand to the cardiac level. This ischemia is also coupled to an *autonomic derangement* (Kember *et al* [1]) where heart

importance is slowly decreased to a pre-determined and randomized minimum while the blood importance is simultaneously slowly increased to its maximum value of one. The response of twin networks to this scenario is found to be strongly dependent upon the rules by which decisions are made to update neural activities. In this way we show that twins may diverge in their response to exactly the same pathology. This observation of divergence is due to the way in which the infinite set of steady-state solutions and the pathway toward a steady-state depends upon event-driven rules used to update neurons. The divergence represents a form of singular behavior which renders the average response of a population meaningless because it is not possible to link the ensemble population response to the individuals within the population. To examine this concept of *twin divergence* we examine how the network responds to sequential simulation versus event-driven simulations, as described above, with and without the ischemic event.

1.3 Literature Review

Networked cardiac control as presented in Kember *et al* latest work [1] was implemented as a sequential program and it is the focus of this thesis to qualitatively investigate what happens when such a model is implemented as an event-driven simulation. Such an investigation will allow for the observation of how orders of operations affect the results of the model simulation. The fundamental difference between a sequential simulation and an event-driven simulation, in our case, is that the sequential simulation is generalized to include an explicit mathematical model of a causal chain that determines the path by which a model evolves. The area of event-driven simulation is clearly full of open and deep mathematical questions and has only begun to be considered. The intent is to limit this thesis to a purely qualitative investigation focused on the possible significance of event-driven simulation for modeling of cardiac control. The question examined here is: "what are some implications of event-driven programming to simulation of the cardiac neural network model presented in Kember *et al* [1]?" There are a number of languages written for event-driven programming. This thesis uses the Visual Tag System (VTS) language developed at Trihedral Engineering Limited, Bedford, N.S. to implement our event-driven simulations.

A literature review was conducted on event-driven simulations and since these simulations are in the early development stages they tend to be spread over a broad range of topics and mostly dedicated to specific applications. For this reason, the literature discussion presented in this thesis will drift away from cardiac control in order to gain a perspective of what is being done in the event-driven area.

The main thread that drives much of the research presented as a literature review in this thesis into event-driven simulation is the desire to improve computational efficiency. Improving computational efficiency is desirable for event-driven simulation because significant computational resources are utilized when executing this type of simulation due to the large number of calculations and events within the simulation. In addition, because there is a large computational expense associated with imposing mathematical models designed to follow causal chains, there is interest in the academic community to study event-driven simulation. For example, in the cardiac model considered in this thesis from Kember *et al* [1] (which will be described in detail in the next subsection), each update of the neural network involves the update in activity of a single neuron. The neuron that is to be updated is the one that best satisfies a set of criteria based on the mathematical model for causality. Because the neural updates are based on such a set of updating criteria, there is a confusing assortment of qualitative methodologies that seek to reduce computational time while respecting the parameters or constraints of a particular application. In other words, researchers make approximations to the physics of their applications mainly in an attempt to reduce computational time. To simplify the discussion of the literature, methods for the implementation of event-driven simulation are used for categorization. Any insights gained with respect to an application are also noted wherever possible.

Systems composed of particles can be modeled as hard rigid bodies; therefore, event-driven simulations are a natural fit due to the fact that modeling these particles imposes an event-based criterion to update the simulations in each time step. Within particle physics *event-driven molecular dynamics* (EDMD) have been used to simulate molecular dynamics. For example, research is presented for optimizing event-driven simulations

(Michele [2], Valentini and Schwartzentruber [3], Lerner *et al* [4]) where rules defining event-driven causality are implemented through *nearest neighbor lists*. For the research presented in Michele [2], they define three quantities: (i) collision neighbors that are added to or subtracted from a neighbor list based on a criterion, (ii) collisions between particles are simulated, and (iii) cell-crossing with rigid bodies is assessed. Cell-crossing is the activity in which particles interact (but not collide) in EDMD models. The key point these researchers (Michele [2], Valentini and Schwartzentruber [3], Lerner *et al* [4]) use is the notion of event time and calendar time. These two types of time are explicitly separated in the simulation code and the decision to update either of these flows from (i) to (iii) is made based on a set of global causal criteria.

A spiking neural network (Caron *et al* [5]) is implemented in an event-driven manner in hardware with an analog integrated circuit model representation. Calendar and event time are considered in Caron *et al* [5] research as well, where the number of events that occur within a unit of calendar time are found from the processing queues at a given rate [5]. In the work presented in Caron *et al* [5], the neurons within the model that are subject to the event-driven model have their activities updated based on a queuing algorithm for heaped stacks. Neurons within the network are represented as graphical processing units (GPUs) and digital signal processors (DSP) that are integrated to form an analog very-large-scale integrated (AVLSI) circuit. The neurons have *synaptic connections* within the AVLSI circuit model and it is the state and condition of these connections that are subjected to event-driven updating.

A comparison of sequential programming and event-driven simulations is made in Caron *et al* [5]. The sequential implementation is a series of programming steps where a neuron's state is updated using a sequence of operations that is entirely independent of the neuron states within the network. On the other hand, in the work presented by Caron *et al* [5], event-driven simulation is presented with rules where any update of a neuron's state only occurs for the neuron that best satisfies the rules. These two aspects of sequential and event-driven programming are analogous to the approach followed in this thesis. The main conclusion in Caron *et al* [5] is that hardware implementation of spiking

networks may be a candidate for implementing event-driven neural updating using hardware that uses less computational time. Caron *et al* [5] do not consider a closed-loop control system as is done here but their work could certainly be extended to control applications.

Other attempts to reduce computational time involve combining the molecular dynamics ideas [2] and *look-up table* approaches. An example of this strategy is an event-driven model of granule cells in the brain presented by Carrillo *et al* [6]. They use a simulation method dubbed the *event-driven lookup-table-based simulator*. This approach makes use of pre-determined, smaller networks to approximate the simulation of large-scale networks. At the start time of the simulation, cell states are separately updated using a system of about 15 coupled differential equations for each individual cell. Then, each cell is placed within a small-scale, local network that is represented by another set of fewer differential equations (thus an approximation). The small-scale networks are then updated using the current state of each cell inhabiting the small-scale network, which is previously derived. Finally, the event-driven aspect is implemented by comparing the small-scale network results with previously-computed small-scale network behaviors that form a linear combination of these results. In other words, the results obtained for simulating the single cell are used to build a system of small-scale networks that are then used (as an approximation) to scale-up into the global, event-driven model. Hence, this methodology should be considered an approximation.

The accuracy of hybrid methods was also estimated by comparing simulations with and without the use of look-up tables. Their [6] results found varying accuracy exposing a strong dependence of the look-up table accuracy on the precise details of the network. The main benefit, and possibly a window into understanding where and how look-up tables may work, is that the computational time depends only on network activity and not the network size. In other words, less active neurons are assumed to exert a smaller impact on the network state. This observation runs contrary to biological applications where, for example, low-level activity which tends to enable plasticity (high-level activity tends to saturate plasticity) has been found to be extremely important to adaptive

control (Prinz *et al* [7]) and this could lower the usefulness of look-up table approximations.

Another example of research into look-up tables is that which is presented in Roeller *et al* [8] where an event-driven sinusoidal shaking simulation is used to model the separation of granular material and powders into various sizes. These models are typically treated by using differential equation formulations that treat the powder or granular material as a viscous fluid. This resembles the work completed by Carrillo *et al* [6] and Roeller *et al* [8], who use look-tables to approximate event-driven simulations. Carrillo *et al* [6] and Roeller *et al* [8] work is of interest due to the unique approach used to model the velocity and trajectory of movements of the particles within the powder matrix. The events in their simulations are the collisions whose position and timing are determined using the event-driven aspect [8]. To improve the simulation time, the algorithm for event-driven simulation is sped up using look-up tables. Like in other references [2-6, 8-14] cited in this section, the tables are used to store other, already-calculated, outcomes in the simulation. The simulation time using the tabulation method is constant (compared to other simulations that used the tabulation method) as observed before [6] and is independent of model complexity [8]. However, the accuracy of the tabulation method for storing trajectories versus more standard and validated approaches is not provided.

Other researchers consider look-up table approaches, and in Girardi [9] spherical insulating grains interacting with an external electrical field with application to storage of grains, transport, painting and separation of material. Carrillo *et al* [10] have also extended the study of event-driven look-up table approximations to simulate large-scale neural populations. They present similar results to those found in [6]. The premise of the work presented by Carrillo *et al* [10] is that biological neural systems are parallel in nature and that these systems are massive in size and eludes computational capability. Carrillo *et al* [10] also state that the dynamics of these systems are widely open issues since event-driven simulation exposes dynamical aspects not considered before in research. An issue with look-up table approaches is that there is no systematic

examination that can provide guidelines for when these approaches provide accurate results.

An important work is that of Omelchenko and Karimabadi [11]. Their application is in the field of particle-in-cell models with application to physical biology and the study of cell dynamics. Here, they [11] present an alternative to traditional time-step particle-in-cell models, which are very time intensive and require significant computing resources. Their [11] alternative is to use event-driven simulation techniques like those presented in this thesis. The event queuing involves a so-called advance-synchronize-schedule algorithm where cell tasks are synchronized for tasks with vastly different numbers of sub-tasks. This is akin to adapting the speed that robots work on different assembly tasks within a car manufacturing plant. Each robot must work at a different pace in order to allow the assembly line to proceed in a predictable way. Omelchenko and Karimabadi [11] compare results obtained by traditional explicit simulation (time-driven) and the event-driven simulation and obtain good agreement in all cases. A key assumption latent in their algorithm is that their event-driven model ensures global stability by ensuring that all sub-tasks are completed within the required time. The extent to which this sort of rule is present in physiology is unclear.

Event-driven programming (Host *et al* [12]) has also been applied to predict and allow for the mitigation of bottlenecks due to the number of or training level of employees, in industrial processes. The simulation work presented by Host *et al* [12] use state programming to simulate queue times and dynamics in industrial applications.
In Denkena *et al* [13] event-driven programming is used to optimize modern manufacturing practices. The purpose of their work [13] is to present a hierarchical online simulation technique for cost-optimized integration of condition-based maintenance for an industrial assembly line. The primary focus of the model presented in [13] is maintenance planning and the consequences of production loses if equipment is not maintained in a preventative way. The application presented by Denkena *et al* [13] involves a hierarchy in the model, which are features of the work presented in this thesis (see details the next subsection).

Event-driven programming has also found application in the optimization of metropolitan light rail systems in Chile (Grube *et al* [14]). Unlike other event-driven simulations presented in this section, the application presented in [14] is heavily dependent upon a general stochastic model to simulate public transportation dynamics. The main conclusion from their work [14] is that they were able to produce a tool that was successful in accurately simulating vastly complex networks in relatively short periods of computational time. The authors of [14] were able to present a finding that appears to have eluded other researches based on purely differential equation approaches and they showed that using an hourly-dependent holding strategy improves the public transportation system as a whole [14].

In this thesis look-up tables are not used and a full event-driven simulation of Kember *et al* [1] is implemented using Trihedral Engineering's VTS software. These event-driven results are compared with a sequential Java-based simulation. Qualitative details on the mathematical model used in this thesis are provided in the next subsection. Moreover, definitions of key terms are presented as well.

The thesis is organized in the following manner: (i) a quantitative description of the mathematical model is provided in Chapter 2, (ii) details needed to understand precisely how sequential and event-driven simulation are implemented are found in Chapter 3 and (iii) the clinically relevant work involving the comparison between sequential and event-driven implementations with and without ischemia is provided in Chapter 4.

CHAPTER 2 MODEL OF NETWORKED CARDIAC CONTROL

A quantitative description of the model presented in this thesis is provided in this chapter. First, the physiology for control of cardiac output is given. From there, the networked cardiac model specifics are outlined. The model is highly dependent on the heart and network functions. These functions and their fundamental equations are outlined. Lastly, internal network processing features are discussed.

2.1 PHYSIOLOGY FOR CONTROL OF CARDIAC OUTPUT

Most studies of cardiac control are based upon the assumption that the control of heart function is primarily centered in the brain or *central command*, as outlined in various literature references [15-19]. Neural systems related to cardiac control that exist outside central command are believed to function only as conduits for sensory feedback from cardiovascular afferent neurons and as a means to relay control decisions made at the level of the brain to the heart. This central command concept is unable to form a basis for the relationship between dysfunction of the autonomic nervous system and cardiac arrhythmia. Reasons why this is the case are clear from anatomical studies [20-22] that have shown the existence of a wider peripheral control system of heart function.

Therefore, a hierarchical neural network model for control of the heart with three levels and three populations of neurons was introduced [1]. The top level represents a neural population residing in the brain, or central command, the middle represents a neural population residing between the brain and the heart and the bottom represents a neural population residing on the surface of the heart and in fatty tissues surrounding the heart.

This model has been used to explain features of heart rate anomalies such as the presence and absence of so-called Mayer-Wave oscillations in heart rate [23] that were previously beyond the scope of central command ideas. The introduction of plasticity to the network model [24] allowed some insight into the level of dissimilarity that exists among neural activity that is again consistent with common observations. Most recently [1], the

network model was used to explore the response of the network to a cardiac ischemic event (heart attack) and the role of autonomic derangement and network plasticity in determining the response to conditions created by the ischemia. In the following section, the model of networked control presented in [1] is laid out for the reader.

2.2 Networked Cardiac Control Model

The neural network has three levels. These levels, 1, 2 and 3 are also respectively referred to as the bottom, middle and top, respectively, or cardiac, intra-thoracic and central. We use two indices, j and k to identify the k^{th} neuron at the j^{th} level, and the activity of neuron is $0 < s \leq 1$. The three levels have N_1, N_2 and N_3 neurons, respectively.

2.3 The Heart

The neural network receives external information in the form of demand for blood flow and feedback of heart rate. It processes this information and then the level of activity at the cardiac level of the network is used to update the heart rate. The elements of this are described next.

The heart rate is updated at times $t^{(n)}$, $n = 1, 2, \ldots$, based on an incremental move $\Delta M^{(n)}(t)$ constructed from the average activity at the cardiac level

$$\bar{F}^{(n)} = \frac{\sum_{k=1}^{N_1} S_{1,k}^{(n)}}{N_1} \tag{2.1}$$

from which the move (update to the net move defined in equation (2.3))

$$\Delta M^{(n)} = \beta(\bar{F}^{(n)} - \alpha) \tag{2.2}$$

and α is a reference level, β is a positive gain and $S^{(n)}_{1,k}$, $k = 1, 2, \ldots$, is the level of activity of neurons at level 1 (cardiac) of the network at time interval $t^{(n)}$.

The net move is then

$$M^{(n)}(t) = \sum_{i=0}^{i=n-1} \Delta M^{(i)}(t) \tag{2.3}$$

and within the interval $t^{(n)} \le t \le t^{(n+1)}$, $n = 1,2, \ldots$, the heart rate $H^{(n)}(t)$ satisfies the first order, linear, time-invariant differential equation

$$\tau_H \frac{dH^{(n)}(t)}{dt} + H^{(n)}(t) = M^{(n)}(t) \tag{2.4}$$

where τ_H is a process-time constant. In what follows the time variable t shall not be shown explicitly. The heart rate, as stated in Chapter 1, is a scaled variable the runs between zero and one, with zero and one representing the minimum and maximum heart rate, respectively, for a typical adult male. (*i.e.* 45 beats per minute (minimum) and 165 beats per minute (maximum)).The net move is assumed to be constant within each time interval. The heart rate condition applied at the beginning of an interval is equal to the heart rate at the end of the previous interval. In this way, a piecewise, continuous heart rate solution is constructed.

2.4 The Network

This subsection contains details on network processing. The subsection is divided into two smaller subsections, one on network processing of external outputs and the other on network internal processing.

2.4.1 Network Processing of External Inputs

The activity level of neuron j, k in time interval $t^{(n)}$ is $S^{(n)}_{j,k}$ and is affected by three elements: (i) current blood demand, (ii) current heart rate and (iii) the level of activity of neighboring neurons. Changes in neural activity due to these effects are, respectively, $\delta_1 S^{(n)}_{j,k}$, $\delta_2 S^{(n)}_{j,k}$, *and* $\delta_3 S^{(n)}_{j,k}$.

We define two types of neurons: (i) *heart rate neurons* that respond to heart rate and neighboring neural activity and (ii) *blood demand neurons* that are affected by demand for blood flow and neighboring neural activity. The total change in the state of activity of a neuron j, k is therefore given by

$$\Delta S^{(n)}_{j,k} = \delta_1 S^{(n)}_{j,k} + \delta_3 S^{(n)}_{j,k} \tag{2.5}$$

$$\Delta S_{j,k}^{(n)} = \delta_2 S_{j,k}^{(n)} + \delta_3 S_{j,k}^{(n)} \tag{2.6}$$

The importance, weight, or sensitivity that neurons place on heart rate H and blood demand D is, respectively, $h^{(n)}_{j,k}$ (heart importance explained in Figure 1.1) and $d^{(n)}_{j,k}$ (blood importance explained in Figure 1.1) so that the change in neural activity due to these effects is

$$\delta_1 S_{j,k}^{(n)} = -\frac{h_{j,k}^{(n)}}{\overline{h}^{(n)}} \times H^{(n-L)} \tag{2.7}$$

The heart rate $H^{(n-L)}$ is delayed by a process-time constant $\tau_L = L\Delta t$ for the feedback signal to arrive at the network and the average $\overline{h}^{(n)}$ over the entire network is used to scale the heart sensitivity $h^{(n)}_{j,k}$. In the case of sensitivity to blood demand we write

$$\delta_2 S_{j,k}^{(n)} = -\frac{d_{j,k}^{(n)}}{\overline{d}^{(n)}} \times \frac{D^{(n)}}{D_{max}} \tag{2.8}$$

and the demand is scaled with respect to a maximum demand D_{max}.

The *(n)* superscripts show that the sensitivities $h^{(n)}_{j,k}$ and $d^{(n)}_{j,k}$ can change in time. Changes in sensitivity, *i.e.* heart importance and blood importance, to heart rate feedback and external input (see Figure 1.1) will be used as a representation of neural effects exerted by myocardial ischemia. In particular, the pathological evolution of sensitivity to blood demand and heart rate, *i.e.* neural plasticity (Figure 1.1) driven by ischemia will be called autonomic derangement (see Chapter 1). Neural plasticity is differentiated here from network plasticity (see Figure 1.1), which is assumed to refer to plasticity in the neighbor weightings $P^{(n)}_{J,K}$ internal to the network and these considerations will be examined later in the thesis.

2.4.2 Network Internal Processing

Neurons are also affected by the activity of other neurons in the network. Networking is assumed to occur between a neuron and its neighbors at the same level of the neural network and an adjacent level. For example, a neuron at the cardiac level has neighbors at the cardiac and intra-thoracic level, a neuron at central command has neighbors at the

central and intra-thoracic levels, while a neuron at the intra-thoracic level has neighbors' at all three levels.

The synaptic strength between neuron *j*, *k* and each of its neighbors is represented by a weighting $P^{(n)}_{J(i),K(i)}$, (see Figure 1.1 for description on *neural neighbor synaptic connection*) where $i = 1, 2, \ldots, N_{b(j,k)}$ identify the neighbor neural and level indices and the upper case *J*, *K* refer to the neighbor neural and level indices. The superscript *(n)* shows that synaptic strength can change from one time interval to the next and this is what we term network plasticity [1]. The way in which the state of activity of a neuron *j*, *k* is influenced by its neighboring neurons represent networking among neurons and is the quantity δ_3 in Equations 2.5 and 2.6

$$\delta_3 S^{(n)}_{j,k} = \sum_{i=1}^{N_{b(j,k)}} (S^{(n)}_{J(i),K(i)} - S^{(n)}_{j,k}) \times P^{(n)}_{J(i),K(i)} \quad (2.9)$$

CHAPTER 3 SIMULATION OF THE MODELED CARDIAC NETWORK

In this chapter we describe two methods used to implement the solution of the equations presented in Chapter 2: (i) networked control of heart rate is simulated in a sequential way using the Java programming language and (ii) using event-driven programming based on the VTS programming language. The two methods of simulation are distinguished in our model by the criteria used to choose a neuron whose activity will be updated at each time step. In the sequential Java simulation, a neuron is randomly chosen with replacement for updating of its activity and this is entirely independent of its activity level. On the other hand, neurons are chosen for update in the VTS simulations based on changes in either their activity level or that of their neighbors. The changes in activity that are used to choose a neuron to be updated or to have its activity *interrupted* are called *events* and, therefore, this way of choosing neurons to update is called *event-driven* simulation.

3.1 SEQUENTIAL SIMULATION USING JAVA

The main program pseudo-code is provided directly in the thesis for the Java implementation. A flowchart showing the method of execution for the Java simulation is provided in Figure 3.1. Equations 2.1–2.9 from Chapter 2 are noted within the Java pseudo-code and appear below, starting at line 64 in the code. Similarly, the lines of Java pseudo-code presented below and Equations 2.1–2.9 from Chapter 2 are both noted in the flowchart in Figure 3.1 for cross-referencing.

The code given below is broken into an initialization sequence where objects required for the simulation are constructed between lines 1–19. The simulation takes place over the desired timeframe after line 19.

```
1      public class Controller {
2          public static void main(String[] args) throws IOException {
3        // build filestreams
4        // set simulation variables
5        // construct net object
6        Net myNet = new Net(NumberOfLayers, NumberOfneuronsPerLayer,
7        myNetFilename, Hierarchy);
8        // build heart object
9        Heart myHeart = new Heart(Rate, MaxHeartRate, TimeConstant,
10       myHeartFilename);
11       // build blood object
12       Blood myBlood = new Blood(Demand, MaxDemand, Delay,
13       TimeUntilNextBloodDemand, LastTime, LastTimeN, Noise,
14       myBloodFilename);
15       // build move object
16       Move myMove = new Move(LastMove, NewMove, LastDuration, NewDuration,
17       MinMove, MaxMove, MinNetMove, MaxNetMove, Threshold, NetMove,
18       myMoveFilename);
19       // execute code
20       for (CurrentTime=0; CurrentTime<MaxTime;
21        CurrentTime=CurrentTime+DeltaT){
22        ////////////////////////////////////////////////////////////
23        // floatNetwork = false to TURN OFF plasticity
24        boolean floatNetwork = true;
25        if (floatNetwork) {
26         for (int iTime=0; iTime<3; iTime++) {
27          myNet.operateNet.updateNeighbourImportance...
                                                  RandomAcrossLayers(myNet);
28         }
29         // update neighbour importance to maintain unit average
30         // neighbour importance
31          for (int i=0; i<NumberOfLayers; i++) {
32           myNet.operateNet.operateLayer.update...
                          NeighbourTotal(i, myNet.Layers); //each neuron
33           myNet.operateNet.operateLayer.updateTotalNeighbourImportance...
                                       (i,myNet.Layers); // each Layer
34          }
35          // update network neighbour importance
36          myNet.operateNet.updateAverageNeighbourImportance(myNet);
37          myNet.operateNet.updateNeighbourRenormalization(myNet);
38        }
39        // myocardial ischemia
40        // set infarctNetwork = false to TURN OFF infarct
41        boolean infarctNetwork = false;
42        if (infarctNetwork) {
43         for (int iTime=0; iTime<UpdatesForADeltaT; iTime++) {
44          if (CurrentTimeN >= 50000) {
45           for (int i=0; i<NumberOfLayers; i++) {
46           myNet.operateNet.operateLayer.updateHeartImportance...
                                      (i, myNet.Layers); // each neuron
47           myNet.operateNet.operateLayer.updateBloodImportance...
                                      (i, myNet.Layers); // each neuron
48           myNet.operateNet.operateLayer.updateTotalHeartImportance...
                                       (i, myNet.Layers); // each Layer
49           myNet.operateNet.operateLayer.updateTotalBloodImportance...
                                       (i, myNet.Layers); // each Layer
50           // update Network
51           // compute average HeartImportance across Layers
52           myNet.operateNet.updateHeartImportance(myNet);
53           // compute average BloodImportance across Layers
54           myNet.operateNet.updateBloodImportance(myNet);
55           }
56          }
```

```
57          }
58         }
59         // updateNetwork
60         // for each deltaT simulation increment
61         // we update neighbours 3 times
62         for (int iTime=0; iTime<3; iTime++) {
63          // combine neighbour input and feedback
64          // EQUATIONS (2.5), (2.6), (2.7), (2.8), (2.9)
65          myNet.operateNet.applyNeighbourInputAndFeedbackAcrossLayers...
                                (myBlood, myHeart, myNet, CurrentTimeN);
66         }
67         // update new move EQUATIONS (2.1) (2.2)
68          myMove.operateMove.getNewMove(myMove, ...
                                 myNet.Layers[NumberOfLayers-1], myHeart);
69         // update net move EQUATION (2.3)
70         myMove.operateMove.updateMove(myMove);
71         // update heart rate EQUATION (2.4)
72         myHeart.operateHeart.getNewRate(myMove.NetMove, ...
                               myMove.NewDuration, myHeart, CurrentTimeN);
73         // update new demand
74         myBlood.operateBlood.getNewDemand(myBlood, myHeart, ...
                                CurrentTime, CurrentTimeN, DeltaT, myNet);
75         // outputResults
76         // close files
77       }
78      }
```

3.2 EVENT-DRIVEN SIMULATION USING VTS

The Visual Tag System (VTS) implementation is described in the following sections. All VTS codes are written using the event-driven language developed by Trihedral Engineering, Bedford, N.S. Note that *visual tags* refers to visual software used to build event-detection programs sold by Trihedral Engineering to industrial clients. Such visual software is built using the underlying event-driven language developed by Trihedral Engineering. In our case, we are using the underlying VTS language to encode our biological simulations. When we refer to our code as a *VTS* program it does not mean we are using the *visual tags* software, but are using the underlying event-driven language. Flowcharting and pseudo-code are included below, along with aspects of the VTS language used in this work.

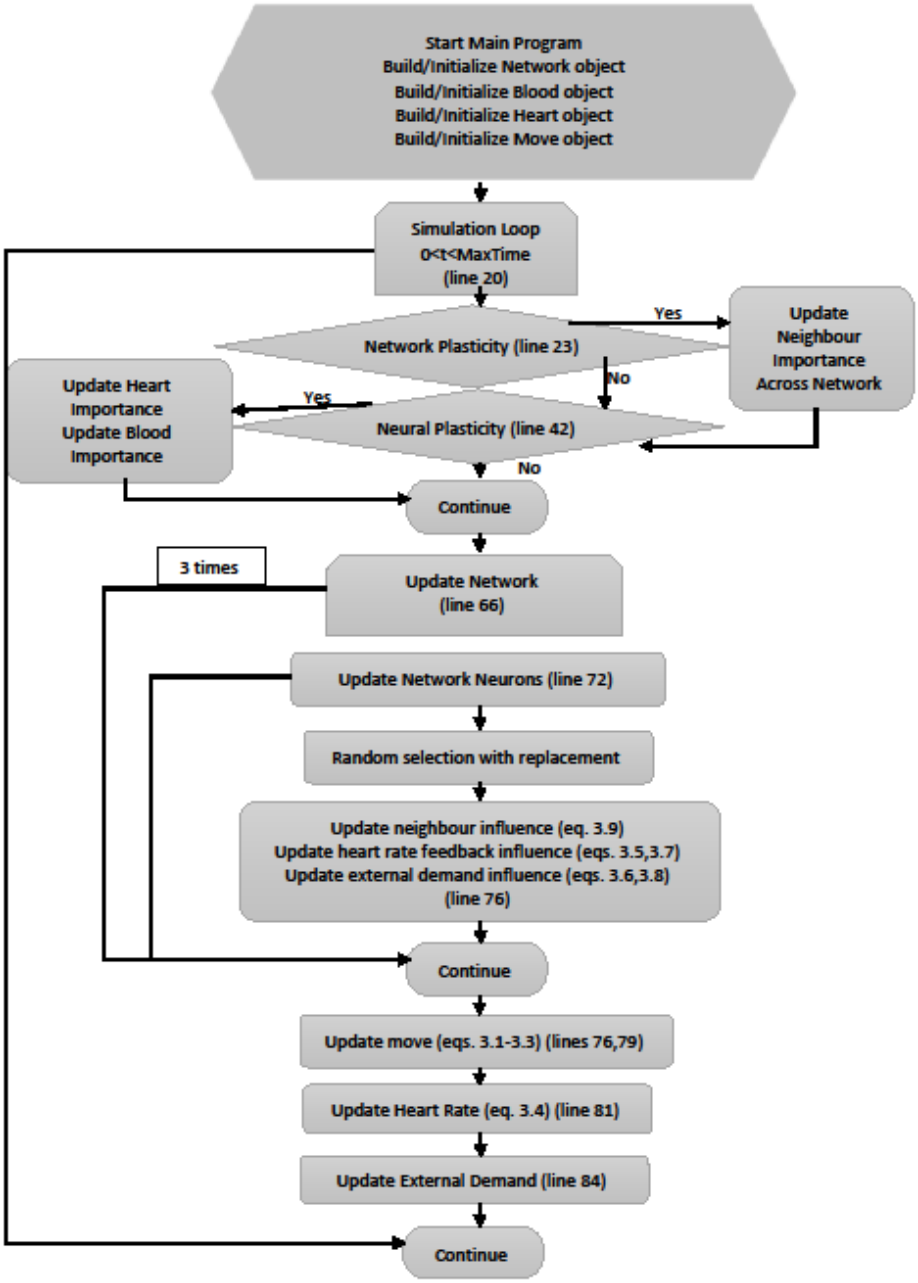

Figure 3.1: Flowchart showing the Java program execution strategy. This is the sequential method for solving the heart control problem. Note the random selection with replacement of the neuron to be updated.

3.2.1 Event-Driven Simulation of Networked Cardiac Control

In our simulation model for networked cardiac control, we used the following *.SRC* files using the VTS programming language:

1. AppRoot.SRC
2. Heart.SRC
3. Net.SRC
4. Layer.SRC
5. Neuron.SRC

Each of these *.SRC* files are explained in succession. Some code fragments are left in syntactical form while other codes are explained in pseudo-format to save space and increase clarity in the thesis. Definitions of terminology are provided in the preamble and line numbers are referenced within each section inside the sub-sections labelled *AboutModuleName.SRC* where *AboutModuleName* refers to the specific module being described. The network that is simulated in VTS has exactly the same structure and properties as that described in Chapter 2 and simulated in Java.

3.2.2 AppRoot.SRC

This module begins by launching the network module (initialize and assign to a pointer *objNet*) followed by launching the heart module (initialize and assign to a pointer *objHeart*).

A module *launch* is like running the constructor for an object in an object-oriented language, like Java. The primary difference is that the timing of the launch is entirely determined by the event-handler that interrupts a state based on the satisfaction of an *if* statement such as those on lines 18 and 25, etc. These *if* statements are called *steady-state* because they are always active in the simulation. In other words, if their condition ever becomes *true*, then the module will enter that particular state.

In our program, initial access to all states in this module are setup so that the launching is forced to proceed in a sequential way even though the event handler keeps testing each state's *if* condition to see if it is satisfied. In this way we ensure that the program runs and proceeds from the network to the heart modules in sequence. Such sequential behavior is

important at start-up since data related to the network structure is read from an input file that is accessed sequentially.

Once the network and heart modules are launched, the A*ppRoot* module enters the *main* state and remains there until the simulation is ended by a sub-module.

A few things to note about this module are:

1. *AppRoot* is the application root module. In a sequential-programming sense it would be the main program. However, here it simply refers to the nature by which variables are *scoped.* In other words, the tree that represents the hierarchy of modules starts with the root. In our VTS program that means the *Net* and *Heart* modules have the *AppRoot* as their parent module so that pointers associated with them, respectively *objNet* and *objHeart* in lines 20 and 28 may be accessed from anywhere within the module tree via *objNet* and *objHeart*.
2. No variables are typecast except for modules. That is why only the *Net* and *Heart* modules are typecast in the variable section that starts on line 3.
3. The entry into the *first state* is commented on line 16 is guaranteed when the module is initially launched since the first state encountered with an *if 1* is executed. Modules only have a single *if 1* set aside for initialization and normally this is placed at the top of a module for clarity. This feature is used in all of the model modules.
4. The *if 1 InitHeart* statement is a condition that is automatically true when the module is launched. Line 20 is executed and then the module state switches from *InitNet* to *InitHeart*.
5. The conditional entry to *InitHeart* on line 28 cannot occur before the variable *NetIsInitialized* is set to true in the Net.*SRC* module (Section 3.2.4) on line 25. In a sequential program this point would be trivially implemented since the lines of code are executed in a systematic way. However, every condition associated with entry to a new state in the module is tested *simultaneously*—in practice this means that the *if* conditions are checked in some fashion dictated by the VTS event-handler. Hence, we obtain a sequential-like result where satisfaction of *if*

conditions leads to a systematic result, *i.e.* we launch the network first, followed by the heart. This approach allows us to control where the VTS is sequential and where it is event-driven and thus control the difference in execution between Java and VTS.

6. This module enters a final state called *Main* where it remains because it has no *if* condition, *i.e.* a state with no condition in VTS is a state that cannot be exited once it is entered.

```
{======================= AppRoot ================================}
{================================================================}
{ VARIABLES }
[
  Net Module "Net.SRC"                  { Launches layers          };
  Heart Module "Heart.SRC"              { Updates heart rate       };
  objNet                                { Handle of objNet         };
  objHeart                              { Handle of objHeart       };
  System                                { access to system library };
  ReadyToRun                = 0         { Full controller is ready };
  NetIsInitialized          = 0         { Net is initialized FALSE };
  HeartIsInitialized        = 0         { Heart is init FALSE      };
  Duration               = 0.01         { Simulation Deltat (secs) };
  MaxNetDurationIndex    = 100010       { Maximum time index       };
]
{ FIRST STATE: InitNet: If '1' is TRUE when code starts }
InitNet [
  If 1 InitHeart;
  [
    objNet = Launch("Net",Self(),Self());{ launch network           }
  ]
]
{ SECOND STATE: InitHeart: state after NetIsInitialized is TRUE     }
InitHeart [
  If NetIsInitialized EverythingIsInitialized;
  [
    { assign a pointer to the heart called objHeart                 }
    objHeart = Launch("Heart",Self(),Self());{ launch heart         }
  ]
]
{ THIRD STATE: entered If NetIsInitialized & HeartIsInitialized     }
EverythingIsInitialized [
  If (NetIsInitialized & HeartIsInitialized) Main;
  [
    ReadyToRun = 1;{ signal that we are ready to run the network    }
  ]
]
{ FOURTH STATE: Last state where main code remains                  }
Main [
]
```

3.2.3 Net.SRC

The Net.*SRC* module is used to build an identical network structure to that used in Java. The network neighbors' choices, their weights, all network parameters, neuron initial states and other quantities are read in using this *.SRC* file.

Features of this module include:

1. The *Net* module is dependent on sub-modules Layer and Neuron as shown in lines 4 and 5.
2. Variables that belong to a *parent* module are referenced by *scoping*. Note that on line 25 the variable belonging to the *AppRoot* module is referenced with a backslash. The backslash means that VTS searches for the variable *NetIsInitialized* starting from the *AppRoot* module. If this variable is not found there, then VTS searches down through the module tree until it finds the variable. To avoid unintended consequences scoping is best done in a direct sense, *i.e.* the full path through the modular tree is specified to reach the desired variable. It is okay to use a single backslash here since *NetIsInitialized* is in the scope of the module *AppRoot*.
3. Just as occurred in AppRoot.*SRC*, the state *if* conditions were setup so that after all layers are launched (pointers are created) the *Net* module is sent to the state *WaitForAllLayersInitialized*. The *Net* module waits in *WaitForAllLayersInitialized* until all of its layers and their respective neuron modules are launched. After that, the *Net* module is sent to the *NetIsRunning* state where it stays. In other words, it cannot exit since there is no condition to do so.
4. There is an *ErrorReading* state on line 18 to detect reading errors by stalling the program. This was used for troubleshooting.
5. Note that VTS has a visual debugger with great flexibility and this allows for very precise debugging and verification of code execution and code properties. Hence, the *ErrorReading* state would be noted in the visual debugger at the code development stage.

```
{============================== Network ==============================}
{=====================================================================}
[
  Layer  Module "Layer.SRC"{ Net Layer module                        };
  neuron Module "Neuron.SRC"{ neuron module                          };
  LayerDictionary{ Dictionary of Layer                               };
  {remaining variable list ...                                        }
]
{ FIRST STATE                                                         }
LaunchLayers [
  If 1 WaitForAllLayersInitialized;
  [
    { read output file from Java for global network parameters        }
    { launch all layers - this uses the Layer.SRC file                }
  ]
]
{ READING ERROR STATE                                                 }
ErrorReading [
]
{ SECOND STATE IF READ IS SUCCESSFUL                                  }
WaitForAllLayersInitialized [
  If (NumberOfLayersInitialized == NumberOfLayers) NetIsRunning;
  [
    { set computed initialized parameters                             }
    \NetIsInitialized = 1;
  ]
]
{ FINAL STATE                                                         }
NetIsRunning[
]
{ End of Net                                                          }
```

3.2.4 Layer.SRC

The Layer.*SRC* module is used to build an identical layer structure to that used in Java by reading them in here.

Some features of this module are listed here:

1. Just as occurred in AppRoot.*SRC* and Net.*SRC*, we set up the state *if* conditions for a layer module so that after its neurons are launched (neuron pointers set aside), the layer module is sent to the state *WaitForAllneuronsInitialized*. Each layer module waits in *WaitForAllLayersInitialized* until all of the network layers and respective neurons are launched and parameters are initialized. Note that even though the neurons are launched, we still have to wait until all of the parameters for the neurons are read in from the input file since the launch only sets up the neuron pointers. After neurons are fully built, the layer module is sent to the

NeuronsAreAllInitialized state where it stays, *i.e.* it cannot exit since there is no condition to do so.

2. This module takes layer parameters in lines 4 and 5: the layer index and number of neurons in layer.

```
1     {============================== Layer ================================}
2     {=====================================================================}
3     (
4       LayerIndex                                  { Layer index         },
5       NumberOfneuronsInLayer                      { neurons in a layer   }
6     )
7     [
8      { variable list }                                                    }
9     ]
10    { FIRST STATE                                                        }
11    Launchneurons [
12      If 1 WaitForAllneuronsInitialized;
13      [
14    { launch neuron module - this runs neuron module.SRC file for each  }
15    { neuron in a layer                                                  }
16      ]
17    ]
18    { SECOND STATE                                                       }
19    WaitForAllneuronsInitialized [
20      If (NumberOfneuronsInitialized == NumberOfneuronsInLayer)
21      n\NeuronsAreAllInitialized;
22      [
23        NumberOfLayersInitialized++;
24      ]
25    ]
26    { FINAL STATE                                                        }
27    NeuronsAreAllInitialized [
28    ]
29    { End of Layer                                                       }
```

3.2.5 Neuron.SRC

The Neuron.*SRC* module is used to build an identical neuron structure to that used in Java by reading in all neuron neighbors, neighbors' weights, etc., and initializing neuron parameters like neuron state to those values used in Java.

Some features of this module are listed here:

1. Each neuron is built in the *Init* state of line 10 and thereafter is sent to the state *NeuronIsBuilt* where it waits until the *AppRoot* module variable *\ReadyToRun* is true.

2. After all neurons and the heart are built, when *ReadyToRun* becomes true the condition in line 21 is executed and the event-handler places each neuron into the *NeuronAwake* state.
3. There are two conditions at the top of *NeuronAwake* and one of them is uncommented depending upon how the simulation will occur: (i) large change interrupt model and (ii) watch interrupt model. Details on these two simulation types are provided later in the thesis.
4. Once all neurons are in the *NeuronAwake* state the simulation begins to *run*. By *run* we mean that whenever the condition for a *NeuronAwake* state to be interrupted becomes true, *i.e.* the interrupt model (large change or watch) becomes true, the neuron script is executed and the neuron state is changed.
5. The interruption of *NeuronAwake* states to update a single neuron state continues until three times the number of neurons in the simulation are updated and then an update of the heart module takes place.
6. Each *NeuronAwake* interrupt to update a neuron state triggers the *NeuronAwake* script commented starting on line 53:
    a. The effect of neighbors on neuron states is imposed starting on line 55.
    b. The effect of heart rate feedback and external demand is noted as a comment on line 64.
    c. Homeostatic and Hebbian plasticity are commented starting on line 75.
    d. Myocardial ischemia (heart attack) is noted starting on line 82 and some lines of code are included thereafter.
    e. Updates of various parameters conclude the script starting on comment line 101.

```
{============================ NEURON ==============================}
{================================================================ ======}
(
  { neuron-specific parameters like index, layer index, ...         }
)
[
{ variable list                                                     }
]
{ FIRST STATE                                                       }
Init [
  If 1 neuronIsBuilt;
  [
   { Read in required information to match Java code network        }
  ]
]
{ ERROR IN READ STATE                                               }
ErrorReading [
]
{ SECOND STATE: GO TO NEURONAWAKE STATE AFTER NET NEURONS LAUNCHED  }
neuronIsBuilt[
  If \ReadyToRun NeuronAwake;
  [
  ]
]
{ FINAL STATE                                                       }
{ ALL neuronS ENTER THIS AWAKE STATE                                }
{ THIS STATE IS INTERRUPTED DURING NETWORK SIMULATION               }
{=====================================================================}
{========================= NeuronAwake State ======================}
NeuronAwake [
{---------------------------------------------------------------------}
{ INTERRUPT NEURONAWAKE STATE IF INTERRUPT MODEL CONDITION IS MET    }
{---------------------------------------------------------------------}
{ LARGE CHANGE INTERRUPT MODEL: COMMENT FOR WATCH INTERRUPT MODEL    }
{ If  (DeltaState > \objNet\MaximumDeltaState * 0.8 &&
       (\objNet\neuronLog/\objNet\TotalNumberOfneurons <
       \objNet\AvgneuronLog))
       ||
      (ItsMyTurn/MaxItsMyTurn > \objNet\WhoIsNext*0.9 &&
      (\objNet\neuronLog/\objNet\TotalNumberOfneurons <
      \objNet\AvgneuronLog));
  [
}
{ WATCH INTERRUPT MODEL: COMMENT FOR LARGE CHANGE INTERRUPT MODEL    }
  If (\objNet\neuronLog/\objNet\TotalNumberOfneurons <
     \objNet\AvgneuronLog) &&
   Watch(1,{watch for ANY change in <=90 neighbours                 }
    WhoInputDictionary[InputKeyList[0]]\neuronState,
    WhoInputDictionary[InputKeyList[1]]\neuronState ...
    );
 [
{---------------------------------------------------------------------}
{ BEGIN NEURONAWAKE SCRIPT                                           }
{---------------------------------------------------------------------}
{ Apply Neighbour Inputs                                             }
 NewInput = 0.0;
 I = 0;
 WhileLoop( I < HowManyInputs,
   NewInput +=
   (WhoInputDictionary[InputKeyList[I]]\neuronState - neuronState) *
    ImportanceInputDictionary[InputKeyList[I]];
   I++;
 );
```

```
64     { Apply HeartRate Feedback and Blood Demand: update NewInput        }
65      NeuronState += NewInput;{ Update Neuronstate                       }
66      NeuronState = Max(MinNeuronState,Min(NeuronState,MaxNeuronState));
67     {------------------------------------------------------------------}
68     { RESET LARGE CHANGE INTERRUPT MODEL PARAMETERS                     }
69     {------------------------------------------------------------------}
70     { Update Abs(Change of State) for this neuron                       }
71     { Update Network Max(Change of State) LARGE CHANGE INTERRUPT MODEL  }
72     { Update \objNet\WhoIsNext = Rand();                                }
73     { Update neuron update counter                                      }
74     {------------------------------------------------------------------}
75     { PLASTICITY COMPONENT }
76     {------------------------------------------------------------------}
77     { Homeostatic Plasticity: apply Homeostatic neurons at high state   }
78     { Homeostatic Plasticity: apply Homeostatic neurons at low state    }
79     { Hebbian Plasticity: apply for Hebbian neurons with high state     }
80     { Hebbian Plasticity: apply for Hebbian neurons with low state      }
81     {------------------------------------------------------------------}
82     { MYOCARDIAL ISCHEMIC EFFECT ON NERVOUS SYSTEM                      }
83     {------------------------------------------------------------------}
84      IfThen( \objHeart\NetDurationIndex > 50000,
85       IfThen( HeartRate && HeartImportance > MinimumHeartImportance,
86        Amount = 0.00001*2*Rand();
87           HeartImportance -= Amount;
88            \objNet\HeartImportance -= Amount /
89               (\objNet\TotalNumberOfHeartRateneurons +
90                  \objNet\TotalNumberOfHeartRateBloodDemandneurons);
91         );
92       IfThen( BloodDemand && BloodDemandImportance < 0.99,
93        Amount = 0.00001*2*Rand();
94           BloodDemandImportance += Amount;
95            \objNet\BloodDemandImportance += Amount /
96               (\objNet\TotalNumberOfBloodDemandneurons +
97        \objNet\TotalNumberOfHeartRateBloodDemandneurons);
98          );
99      );
100    {------------------------------------------------------------------}
101    { PERFORM REQUIRED UPDATES                                          }
102    {------------------------------------------------------------------}
103    { Increment neuron Event counter                                    }
104    { put one of this neuron's neighbours into the watch pool           }
105    { END NeuronAWAKE SCRIPT                                            }
106      ]
107    ]
```

3.2.6 Heart.SRC

The Heart.*SRC* module is used to build an identical heart to that used in the Java simulation. *Heart* parameters are set at the top of the module in the variables section that is mentioned on line 4, but not included. The heart takes information about average neuron state at the cardiac level and uses that to construct a new move and update the heart rate.

Some of the features of this module are listed here:

1. The heart is built and the module ends up waiting in *HeartIsBuilt* until the *AppRoot* variable *ReadyToRun* is true.
2. The heart state is interrupted for updating of heart rate when the condition which ensures that three times the number of neurons in the simulation has been updated becomes true.
3. The heart module script is described beginning on line 37 where the move, heart rate and external demand are all updated. Various quantities are sent to the output file (comment on line 44) and variables such as number of neuron updates (number of *NeuronAwake* interrupts) are reset to zero (comment on line 49).

```
{============================== Heart ================================}
{=====================================================================}
[
  { variable list ... }
]
{ FIRST STATE                                                         }
LaunchHeart [
  If 1 WaitForHeartInitialized;
  [
{ heart rate variables                                                }
    Rate = New(\MaxNetDurationIndex+1);
    Rate[0] = 0.0;
  ]
]
{ SECOND STATE                                                        }
WaitForHeartInitialized [
  If !\HeartIsInitialized HeartIsBuilt;
  [
    \HeartIsInitialized = 1;
  ]
]
{ THIRD STATE                                                         }
HeartIsBuilt [
  If \ReadyToRun HeartIsRunning;
  [
  ]
]
{ FINAL STATE                                                         }
HeartIsRunning[
{---------------------------------------------------------------------}
{ ONLY ENTER THIS STATE IF 3xNumberOfNetworkNeurons UPDATES COMPLETE}
{---------------------------------------------------------------------}
 If \objNet\neuronLog/\objNet\TotalNumberOfNeurons >=
                                          \objNet\AvgNeuronLog;
[
{---------------------------------------------------------------------}
{ START HeartIsRunning SCRIPT                                         }
{---------------------------------------------------------------------}
{ update global time index                                            }
{ update move                                                         }
{ update Heart rate using NetMove                                     }
{ update blood demand                                                 }
{---------------------------------------------------------------------}
{ WRITE OUTPUT TO FILE                                                }
{---------------------------------------------------------------------}
{ Update Neighbour Importance so Network Average is unity             }
{ Compute and write quantities for output to external file            }
{---------------------------------------------------------------------}
{ RE-INIT BEFORE NEXT NETWORK UPDATE                                  }
{---------------------------------------------------------------------}
{ Decide if it is time to stop the simulation                         }
{ re-initialize for next set of 3 X NumberOfNetworkNeurons updates  }
{---------------------------------------------------------------------}
{ END HeartIsRunning SCRIPT                                           }
{---------------------------------------------------------------------}
 ]
]
{ End of Heart                                                        }
```

3.3 Comparing Sequential and Event-driven Simulation

Differences in the simulation of networked cardiac control via sequential (Java) and event-driven (VTS) programming are qualitatively available from the flowcharts for Java in Figure 3.1 and for VTS in Figure 3.2. The Java and VTS flowcharts share the same *build* section at the top of their respective flowcharts that ensures that they have the same network structure and initial conditions. Each of the Java and VTS flowcharts also indicate that the network neurons are updated an average of three times before the heart rate is updated. Specifically, at the beginning of each time-step of duration $\Delta t = 0.01$, the network is updated 270 times (three layers, number of network neurons is 90) using Equations 2.5–2.9. After that, the heart rate is updated using Equations 2.1–2.4.

The key and only difference between the two forms of simulation is the choice by which we choose to update the neurons. In the Java simulation a neuron is simply chosen at random with replacement and this is independent of network events (identical to the sequential simulations that were used in [1]). In the VTS simulation a neuron is chosen based upon whether or not its state should be interrupted because it satisfied a criterion for interruption. This is paramount for understanding the difference between the two simulation methodologies. This difference is apparent in the middle of the flowcharts where the Java code has no condition for choosing a neuron for update while the VTS flowchart shows the condition by which a state is to be interrupted (see Figure 3.2).

Differences in the results of simulation using these two methods are the subject of the next chapter; however, more details regarding the comparison between sequential and event-driven program execution are provided next.

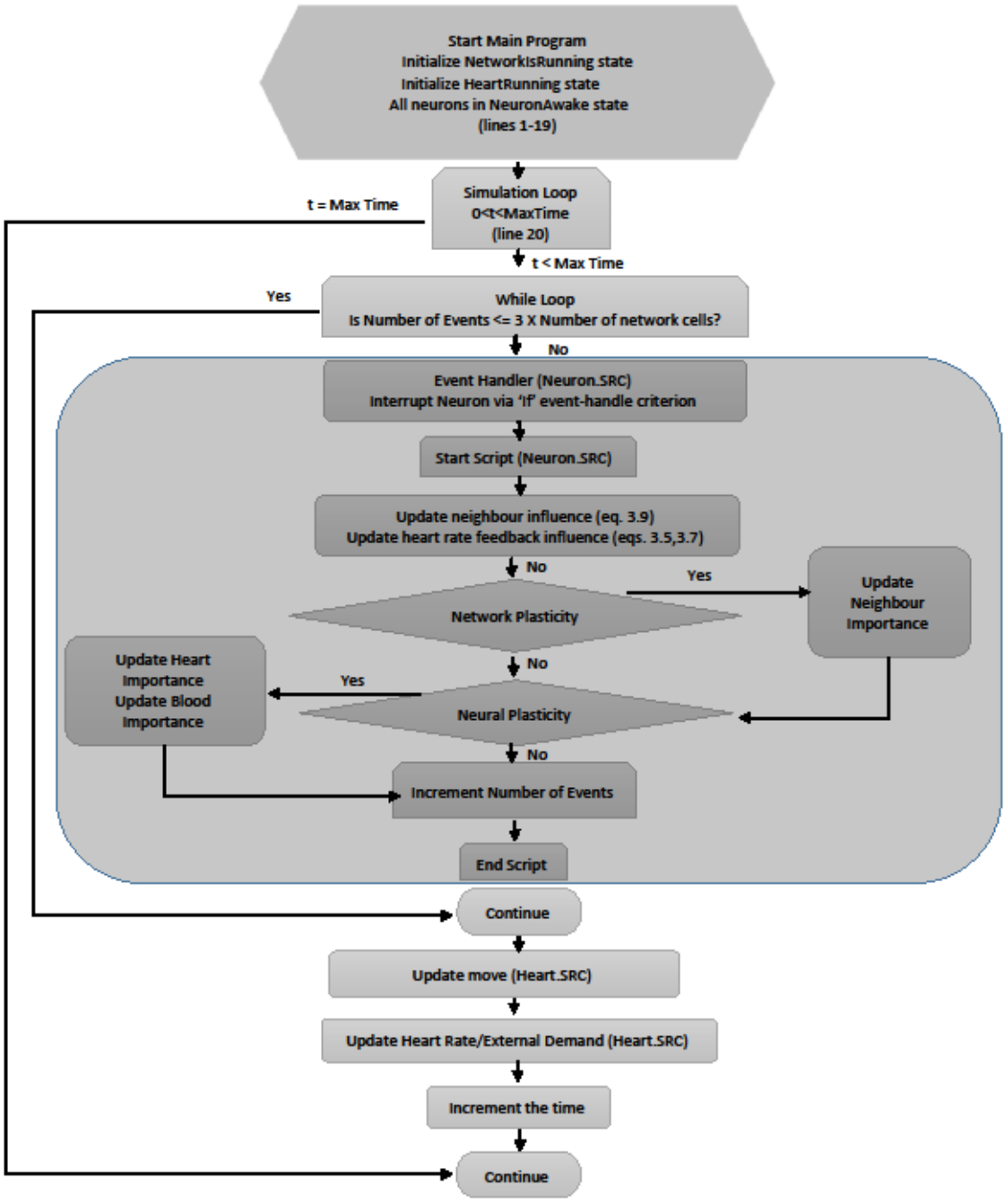

Figure 3.2: Flow chart showing the VTS program execution strategy. This is the state-by-state update method for solving the heart control problem. The state-by-state calculation updates are present inside the larger grey box in the figure.

To demonstrate similar simulation results between Java and VTS, a total of six simulations were executed: three using Java and three using VTS. Blood demand varied from 0.1 to 0.5 using an increment of 0.2 for each model. The total simulation time for all simulations was 200 seconds with a time-step increment of 0.01. This meant the total amount of data points collected for each simulation was 20,000. At the t = 100 s in each simulation, the blood demand was subjected to a 50% increase. For example, for the simulation when the blood demand was 0.1, after 100 s, the blood demand was changed to 0.15 (a 50% increase). With the exception of blood demand, the Java and VTS simulations had the same parameters. To assist in graphing, the data sets were manipulated to align both Java and VTS simulation results (line up peaks, etc.). The results of the simulations are shown in Figure 3.3.

Figure 3.3 is a comparison between Java and VTS heart rate over time of the simulation with a green line showing the change in blood demand half-way through the simulation. There are three plots: the uppermost plot is for blood demand equal to 0.1, the middle plot is for blood demand equal to 0.3 and the bottom plot is for blood demand equal to 0.5. As stated earlier, all simulations had a 50% step-change (increase) imposed in blood demand after 100 seconds had elapsed. For blood demand equal to 0.1 (0.15 after 100 s), the heart rate of both simulation tools is within 0.05 and 0.15 and within 0.1 and 0.2 when blood demand equal to 0.15. For the start of the simulation, both Java and VTS peaks align. However, after about 20 seconds, the peaks begin to misalign. Around 110 s, the peaks align once again; but, after a few more milliseconds, the peaks begin to misalign (the step change in blood demand does not seem to have an effect on the misalignment of the peaks). This suggests the periods of oscillations for both models (Java and VTS) are different. For blood demand equal to 0.3 (0.45 after 100 s), a typical damped oscillation is shown with a steady-state value of 0.30.

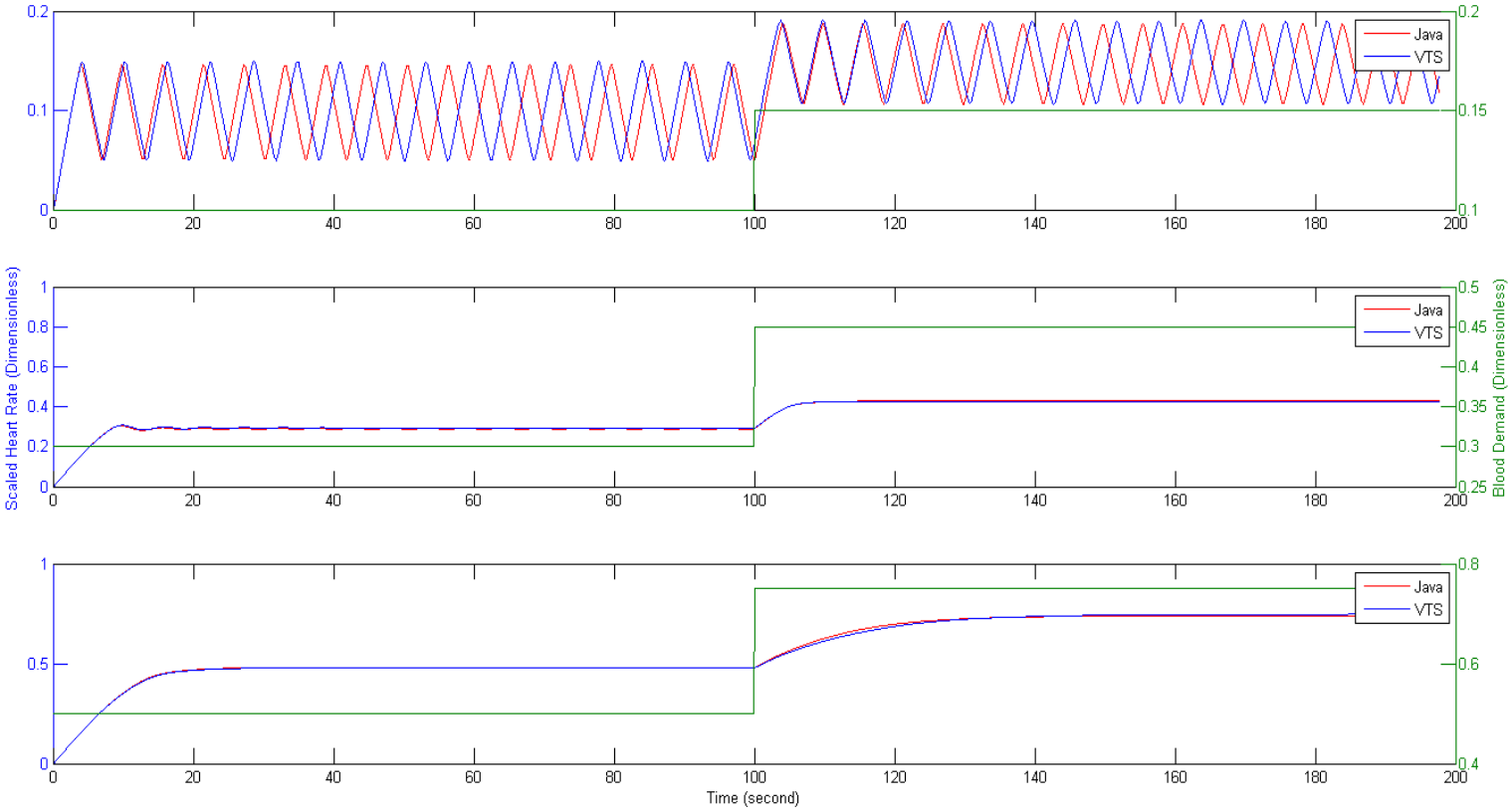

Figure 3.3: Plots showing the numerical comparison between Java and VTS simulations. All plots show heart rate on the left y-axis and blood demand on the right y-axis. Time in s is along the x-axis. In all plots, the Java result is red and the VTS result is blue. The simulation time was 200 s with a step change in blood demand applied after 100 seconds. The top plot uses an initial blood demand of 0.10, the middle plot uses 0.30 and the bottom plot uses 0.50. The Java and VTS simulations show approximately equal amplitude responses regardless of simulation method. The period is also accurately represented in both cases since the time scale over which the discrepancy between the results is resolved is long compared with one.

The Java and VTS results are very similar; however, the Java result seems to oscillate at a slightly lower heart rate compared to the VTS results. After the step-change is imposed, the new system dynamics show a first-order response with no damping, compared to the start of the simulation when blood demand equals 0.3. For blood demand equal to 0.5 (0.75 after 100 seconds), a typical first-order response is shown with a steady-state value of approximately 0.48. The Java and VTS simulation results are essentially the same for this level of blood demand. After the step-change was imposed, the dynamics of the system were similar, showing a first-order response; however, the steady-state value was about 0.75.

In Figure 3.4, Java heart rate is presented on the x-axis and VTS heart rate on the y-axis. For Figure 3.4, three plots are provided: the uppermost plot is for blood demand equal to 0.1, the middle plot is for blood demand equal to 0.3 and the bottom plot is for blood demand equal to 0.5 (all with 50% step-change after 100 seconds had elapsed in the simulation). This is analogous with the damped behavior depicted in the second plot of Figure 3.3. However, after the step-change, the line becomes linear, showing that Java and VTS heart rate are near-equal for each simulation tool. When blood demand is equal to 0.5 (0.75 after 100 s), the line for comparison is entirely linear, which is analogous with the third plot in Figure 3.3, which shows the first-order response. There is a slight change in the line around the point when heart rate is approximately 0.68.

Figure 3.5 shows the same trends as Figure 3.3; however, the time scale is changed to show the heart rate around the time the 50% step-change occurred. The time scale in the plots in Figure 3.5 ranges from 95 to 105 seconds. During this time and at the time the step-change occurred, the heart rate for both simulation tools behaves relatively the same. Recall that the Java and VTS peaks go in and out of phase for the case when blood demand equals 0.1 (0.15 after the step-change).

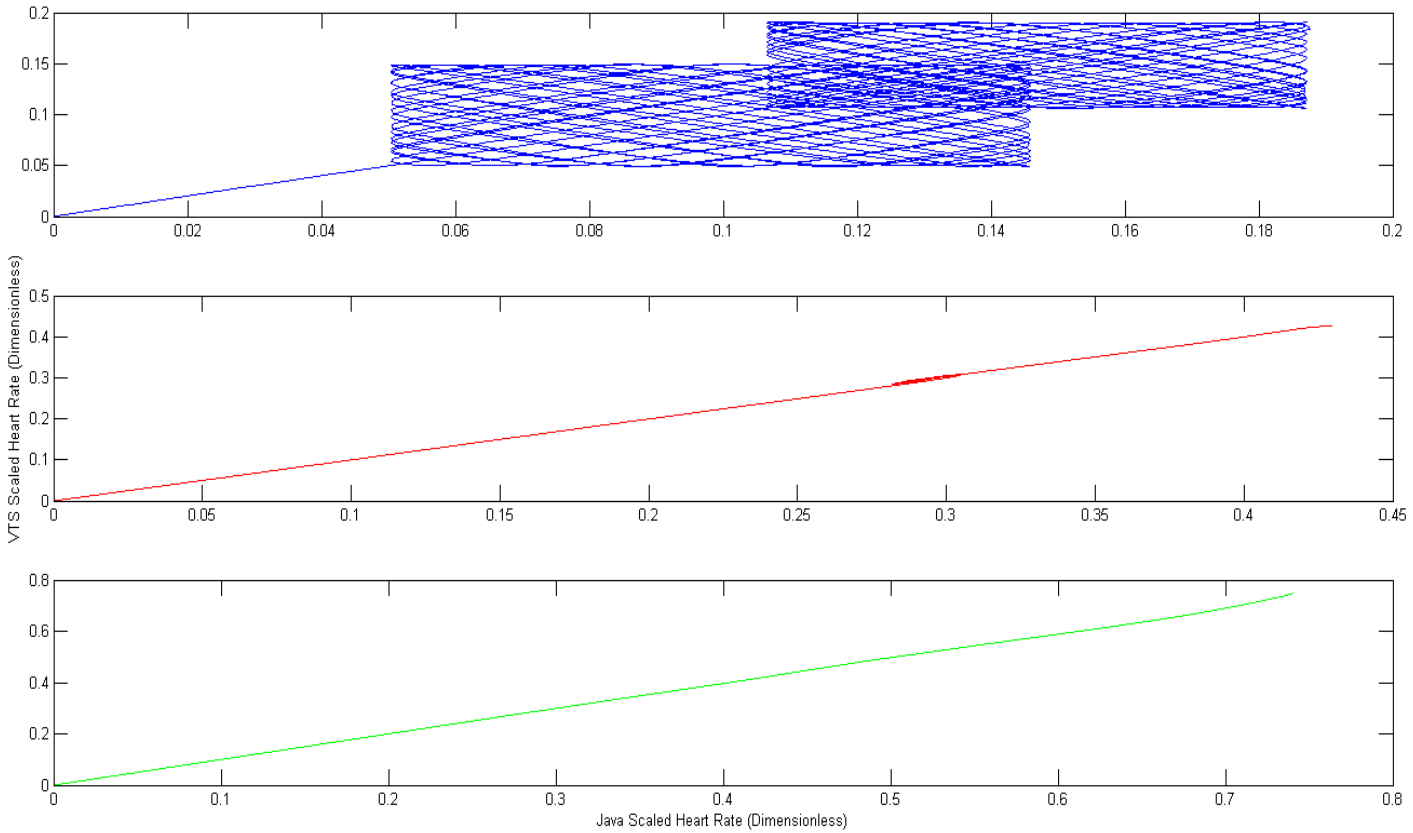

Figure 3.4: Java and VTS simulation comparison. All plots show VTS heart rate versus Java heart rate. The top plot is for the blood demand = 0.10 case, the middle plot is the blood demand = 0.30 case and the bottom plot is the blood demand = 0.50 case. It is observed from these figures that the Java and VTS responses are essential unity in steady state, even with a step change applied at t = 100 s. In addition, altered dynamics due to the event-driven simulation appear as a distinct phase difference in the top plot that is less noticeable for smaller demands in the bottom two plots while in all cases the steady-states are the same in both simulations.

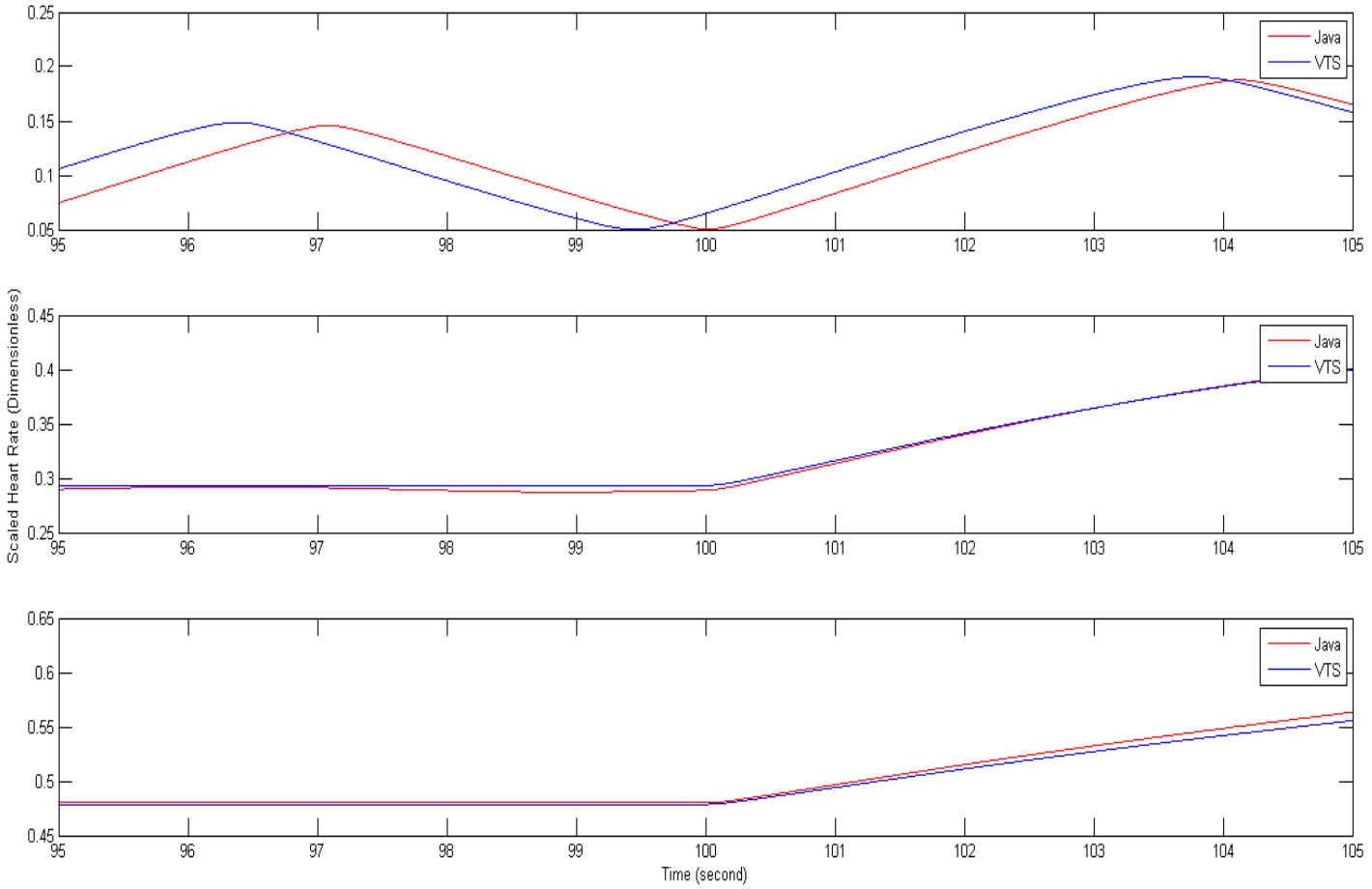

Figure 3.5: Java and VTS simulation comparison. All three plots show heart rate over simulation time (200 s) for Java (red) and VTS (blue) with a step change in blood demand after 100 s. The top plot has initial blood demand at 0.1, the middle plot has blood demand at 0.3 and the bottom plot has blood demand at 0.5. Ignoring the slight time shift in the top plot, the Java and VTS simulation results agree in all three simulations.

CHAPTER 4 ORDERS OF OPERATION IN CARDIAC CONTROL

Our purpose in considering sequential versus event-driven mathematical models is to examine how orders of operation lead to divergence of behavior in nonlinear systems. This is a significant question and we appear to be the first to consider it with respect to biological control (see literature review notes in Chapter 1). We intentionally narrow our focus and only examine this point in a qualitative sense in this thesis with respect to how our model of control of cardiac output is affected.

Both linear and nonlinear systems are dependent upon orders of operation used in computational models. In the case of a linear system possessing a steady-state solution, a stable and consistent numerical approach will lead to the same steady-state within a given order of accuracy regardless of the orders of operations. The nature of the progression to the steady-state solution typically depends upon the order of operations. On the other hand, orders of operations lead nonlinear systems to show differences in the progression to a steady-state and also in the properties of the steady-states that the system converges to. These differences observed between linear and nonlinear models are also seen in our mathematical model for neural-networked cardiac control. A final feature of interest, that again is only explored qualitatively, is the strong differences seen in network behavior that may or may not make their way into significant changes in heart rate.

4.1 SIMULATION OVERVIEW

With the exception of cases where neural plasticity is turned on, all simulations in this thesis are conducted over 1000 second duration. Time is allowed to continue beyond 1000 seconds for simulations where neural plasticity was turned on and this was to observe the different steady-states in the solution (*i.e.* 1000 seconds was not long enough compared to when only network plasticity was turned on). The response of the neural network is considered in all cases for a myocardial ischemia (blood flow loss to the heart tissue, heart attack) that begins midway through each simulation at $t = 500$ s and continues until each simulation ends at $t = 1000$ s. The mathematical model for a myocardial ischemia [1] is assumed to have a greater demand for blood flow that is

primarily directed at the cardiac or lowest level of the control network. Hence, the onset of myocardial ischemia is implemented by doubling the blood demand $D^{(n)}$ at the cardiac level of the network at time t = 500 s and this level is maintained until a simulation ends at t = 1000 s.

The response to the ischemia is considered for situations where network plasticity (synaptic weights are allowed to change) and neural plasticity (sensitivities to heart rate feedback and blood demand are allowed to change) are present in the simulation. These forms of plasticity are outlined below.

4.2 Network Plasticity and Neural Plasticity

We define two forms of plasticity: (i) network plasticity where neighboring weights are changed and (ii) neuron plasticity where the sensitivity of neurons to heart rate feedback and external demand are modified. Network plasticity is a phenomenon that is an operational feature of neurons and because it relates to how the network functions we use the term *network* plasticity. On the other hand, neuron plasticity relates to how each neuron changes its sensitivity to heart rate and external demands.

4.2.1 Network Plasticity

In network plasticity the synaptic strength or *connectivity* among neurons $P^{(n)}_{J(i),K(i)}$ is allowed to change in time. For simulations with network plasticity turned on, this form of plasticity is operational during the entire simulation. The rules by which the change occurs are referred to as *plasticity rules*. To be precise, the actual code fragments used in the neural script component of Neuron.*SRC* in VTS starting near Line 75 of Section 3.2.6 is presented and referred from within the rule statements below. The neighbor weights, or synaptic strengths, between a neuron and its *HowManyInputs* neighbors are stored in the VTS program within the *ImportanceInputDictionary* and these values are modified according to the plasticity rule.

1. Homeostatic plasticity rule: neurons that have a low activity level have their neighbor synaptic weights increased (increases their activity level) while, conversely, neurons that have a high activity level have their neighbor synaptic weights decreased leading to a decrease in their activity levels (line 16).

```
1     { Homeostatic Plasticity; NEURONS MAKING A MINIMAL CONTRIBUTION
                          HAVE THEIR CONTRIBUTION INCREASED }
2         I = 0; { Loop counter }
3         IfThen( Homeostatic && (NeuronState < 0.1) &&
4                                  (SumOfNeighbourImportance < 1.8),
5         WhileLoop( I < HowManyInputs,
6          IfThen( ImportanceInputDictionary[InputKeyList[I]] < 0.9 &&
7             ImportanceInputDictionary[InputKeyList[I]] <
8                                     \objNet\MaxNeighbourImportance,
9           ImportanceInputDictionary[InputKeyList[I]] += 0.00001*0.25;
10              \objNet\AvgNeighbourImportance += 0.00001*0.25 /
11                 \objNet\TotalNumberOfNeurons;
12            );
13            I++;
14          );
15        );
16    { Homeostatic Plasticity: NEURONS MAKING A LARGE CONTRIBUTION
                               THEIR CONTRIBUTION DECREASED }
17        I = 0; { Loop counter }
18        IfThen( Homeostatic && (NeuronState > 0.5),
19          WhileLoop( I < HowManyInputs,
20          IfThen( ImportanceInputDictionary[InputKeyList[I]] >
21                                  \objNet\MinNeighbourImportance,
22       ImportanceInputDictionary[InputKeyList[I]] -= 0.00001*0.25;
23             \objNet\AvgNeighbourImportance -= 0.00001*0.25 /
24                 \objNet\TotalNumberOfNeurons;
25            );
26            I++;
27          );
28        );
```

2. Hebbian plasticity rule: neurons that have low activity have their neighbor synaptic weights decreases which further decreases their activity level (line 1) while, conversely, neurons that have a large activity level have their neighbor synaptic weights increased even further which leads to increased activity (Line 14).

```
1      { Hebbian Plasticity: NEURONS MAKING MINIMAL CONTRIBUTION HAVE
                                    THEIR CONTRIBUTION REDUCED }
2          I = 0; { Loop counter }
3          IfThen( Hebbian && (NeuronState < 0.1),
4            WhileLoop( I < HowManyInputs,
5              IfThen( ImportanceInputDictionary[InputKeyList[I]] >
6                                      \objNet\MinNeighbourImportance,
7               ImportanceInputDictionary[InputKeyList[I]] -= 0.00001*0.5;
8                   \objNet\AvgNeighbourImportance -= 0.00001*0.5 /
9                     \objNet\TotalNumberOfNeurons;
10             );
11             I++;
12           );
13         );
14     { Hebbian Plasticity: NEURONS MAKING A LARGE CONTRIBUTION HAVE
                                  THEIR CONTRIBUTION INCREASED }
15         I = 0; { Loop counter }
16         IfThen( Hebbian && (NeuronState > 0.5) &&
17                                     (SumOfNeighbourImportance < 1.8),
18           WhileLoop( I < HowManyInputs,
19             IfThen( ImportanceInputDictionary[InputKeyList[I]] <
20                                      \objNet\MaxNeighbourImportance,
21              ImportanceInputDictionary[InputKeyList[I]] += 0.00001*0.5;
22                 \objNet\AvgNeighbourImportance += 0.00001*0.5 /
23                    \objNet\TotalNumberOfNeurons;
24             );
25             I++;
26           );
27         );
```

The rules are implemented at time intervals $\Delta t = 0.01$ s. In all cases the change in the activity of a neuron is influenced by a change in the values of its neighbor weightings $P^{(n)}_{J(i),K(i)}$.

4.2.2 Neural Plasticity

This form of plasticity involves changing sensitivity to heart rate feedback and blood demand that is provoked by myocardial ischemia. For this reason neural plasticity begins at time $t = 500$ s with incrementally increasing sensitivities $d^{(n)}_{j,k}$ of blood demand neurons and incrementally decreasing sensitivities $h^{(n)}_{j,k}$ of heart rate neurons throughout the network that continues until the end of each simulation at $t = 1000$ s. The incremental changes Δd and Δh are very small and randomly distributed in the range $[0, 10^{-5}]$. They are implemented at every time step ($\Delta t = 0.01$ s) up to a preset upper threshold of $d^{(n)}_{j,k} = 0.99$ and a preset lower threshold of $h^{(n)}_{j,k}$ which is randomly distributed in the range [0, 0.1].

This form of plasticity is implemented in VTS using the following code fragment with the *NeuronAwake* script:

```
1      { Decrement Heart Importance }
2      IfThen( \objHeart\NetDurationIndex > 50000,
3        IfThen( HeartRate && HeartImportance > MinimumHeartImportance,
4          Amount = 0.00001*2*Rand();
5          HeartImportance -= Amount;
6        );
7      { Increment Blood Importance }
8        IfThen( BloodDemand && BloodDemandImportance < 0.99,
9          Amount = 0.00001*2*Rand();
10         BloodDemandImportance += Amount;
11       );
12     );
```

4.3 SIMULATIONS

As stated above, network and neural plasticity are considered in the presence of a cardiac ischemia. All responses to myocardial ischemia seen in the sequential and event-driven simulations (large change interrupt model and watch interrupt model) are considered in terms of: (i) variations in plasticity, (ii) neural types making up the network population and (iii) degree of neighbor connectedness or number of neighbors connections allowed.

For simulations, we consider the following:

1. Neural and network plasticity were varied:
    a. Network plasticity and neural plasticity are both off.
    b. Network plasticity is on and neural plasticity is off.
    c. Network plasticity and neural plasticity are both on.
2. Neuron types are varied in two ways:
    a. Constant neuron type: all neurons receive both heart rate feedback and external demand.
    b. Variable neuron type: blood demand and heart rate neurons respectively occur in the proportions 1/6 to 5/6 (top or central command), 1/2 to 1/2 (middle or intra-thoracic), and 5/6 to 1/6 (bottom or cardiac). In all cases we are using a network with 30 neurons per level and for the purposes of presentation, the neurons are indexed in the model with: (i) top level having 1 to 25 as blood demand neurons while 26 to 30 are heart rate

neurons, (ii) middle level having 1 to 15 as blood demand neurons while 16 to 30 are heart rate neurons, and (iii) bottom level having 1 to 5 as heart rate neurons while 6 to 30 are blood demand neurons.

3. Neighbor connectedness varied in the following manner:
    a. Maximal neighbor connectedness: Number of neighbors chosen from a peer or adjacent layer is randomly chosen between one and the total number of neurons in the neighbor layer.
    b. Sparse neighbor connectedness: Number of neighbors chosen from a peer or adjacent layer is randomly chosen between one and ten neurons in the neighboring layer. The choice of ten neighbors was made to lead to a qualitatively significant shift in the results. Fewer than ten neighbors leads to results that are somewhat qualitatively related to those seen for ten neighbors.

The blood demand $D^{(n)} = 0.05$ is applied throughout the simulation with the level being doubled to a value of 0.1 at the cardiac level to simulate myocardial ischemia (heart attack). This choice of blood demand leads to oscillatory behavior driven by localized instability [24]. These levels are useful for examination of differences in the response to sequential versus event-driven simulation since they tend to increase the effects of both forms of plasticity.

4.4 Network Plasticity Off, Neural Plasticity Off

The sequential and event-driven simulations are presented for several cases where the differences are most pronounced and which capture the main behaviors. Thus we narrow our focus to the constant neuron type since the variable neuron type has similar results. A discussion of a network with constant neural type is compared for sequential simulation (Java) and event-driven simulation (VTS) for two cases involving the watch interrupt model and the large change model.

4.4.1 Java and VTS Watch Interrupt Model

The heart rate derived from the VTS watch interrupt model (referred to as VTS watch from here on in) and the Java model are presented in Figure 4.1. The heart rate is in close agreement with approximate differences in the amplitude and period of 2–3%.

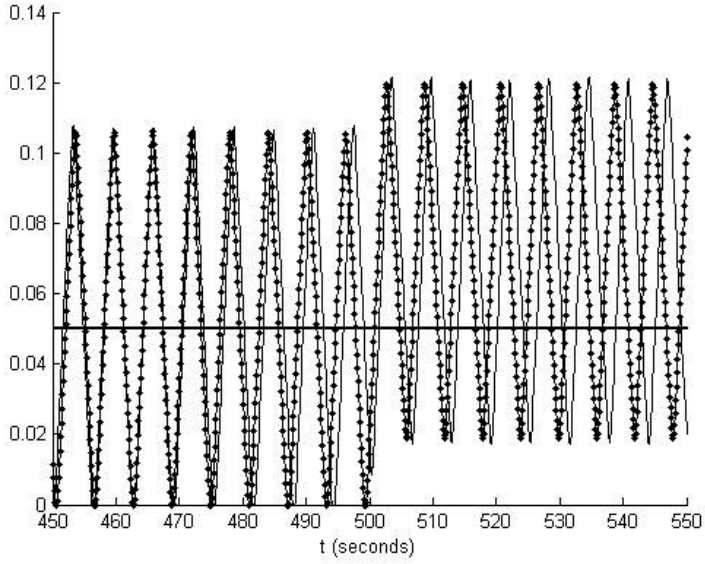

Figure 4.1: Scaled heart rate (dimensionless, on the y-axis) plotted against simulation time for both Java and VTS. VTS heart rate is shown in black dotted lines and Java heart rate is shown in grey continuous lines. Here the watch interrupt model is used for both Java and VTS.

The next point to investigate is whether the close agreement between the heart rate is mirrored in the network processing. To gain insight we consider the number of updates that a neuron experiences per time step during the simulation and we do this in two ways:

1. Raw updates per time step: the number of updates at time *(n)* for neuron *j*, *k* is $U^{(n)}_{j,k}$. This is presented in Figures 4.2 and 4.3. The horizontal axis is the neuron index and is explained directly on the figures. Two things are immediately clear: (i) there is no observable bias in updating neurons in the Java simulation as expected, but there is significant bias within the VTS simulation, and (ii) the VTS simulation also shows less updating of the cardiac level as compared with the middle and upper layers.
2. Average neural update: we can gain more understanding of the differences between Figures 4.2 and 4.3 by considering the number of updates a neuron experiences relative to the total updates with respect to a reference level

$$\overline{U}_{j,k} = \frac{\sum_{i=n_{min}}^{i=n_{max}} U_{j,k}^{(i)}}{\sum_{i=n_{min}}^{i=n_{max}} \sum_{j=1}^{3} \sum_{k=1}^{N_j} U_{j,k}^{(i)}} - \frac{1}{3N} \tag{4.1}$$

where $3N = 90$ is the total number of network neurons and n_{min} to n_{max} is the time index over which the average update is computed. The reference level $1/3N = 1/90$ in Equation 4.1 is the limiting case of each neuron receiving equal updates per time-step on average. Hence, the term $\overline{U}_{j,k}$ is a measure of the deviation from each cell receiving an identical number of updates per time step. Note that we refer to this quantity as an *average* update, for lack of a better term, while bearing in mind that $\overline{U}_{j,k}$ is a fractional update relative to a benchmark. In Figures 4.4 and 4.5 we present $\overline{U}_{j,k}$ and its associated histogram where the columns are $\overline{U}_{j,k}$ in the first column and $\overline{U}_{j,k}$ histogram in the second column. The first row of these figures is before the myocardial ischemia $1 \le t \le 5000$ and the second row is after the myocardial ischemia ($5001 \le t \le 10000$, with the VTS simulation ending slightly earlier).

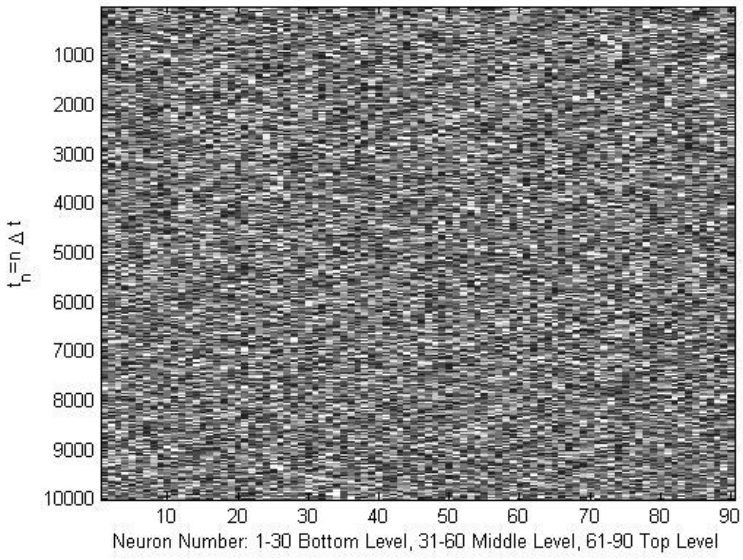

Figure 4.2: Neural network in Java for raw updates per time step. No observable bias towards updating a neuron or a layer in the network.

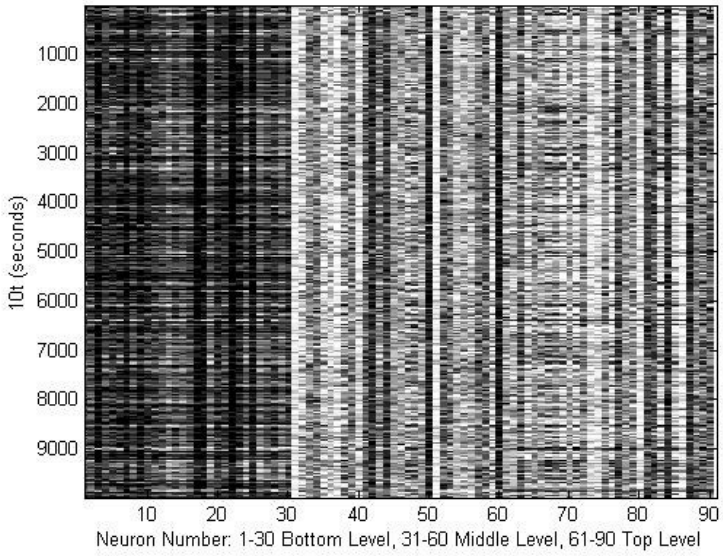

Figure 4.3: Neural network in VTS for raw updates per time step. Observable bias is present here, i.e. the bottom layer shows significantly more update activity compared to the other layers.

The histogram results in Figures 4.4 and 4.5 show the VTS simulation as having two orders of magnitude more variation in updates between neurons than the Java simulation; this is further quantified in the adjacent histogram. When we consider the number of updates that each neuron experiences as the simulation proceeds, it is quite clear that there is quite a difference between the two simulations in spite of their close agreement in heart rate.

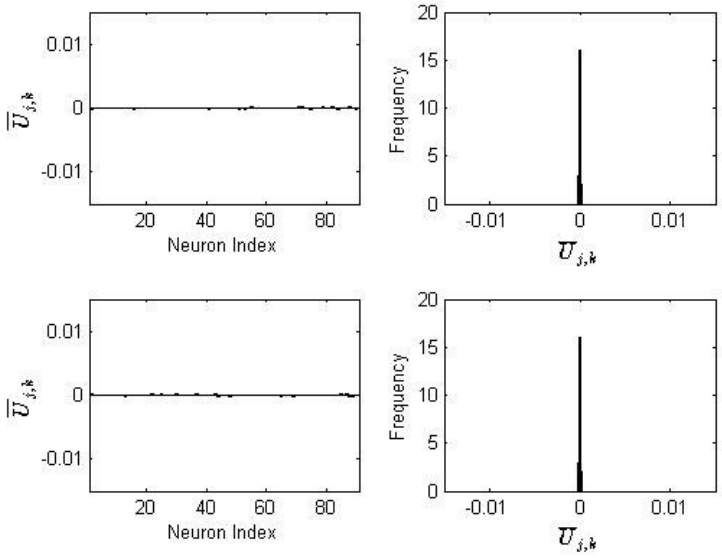

Figure 4.4: Histogram of Java results for average number of neural updates. The first column is the relative number of updates per neuron as defined in Equation 4.1. The second column is the histogram of the relative number of updates. There is minimal variation in the amount of neural updating between as indicated by histogram spike-like nature.

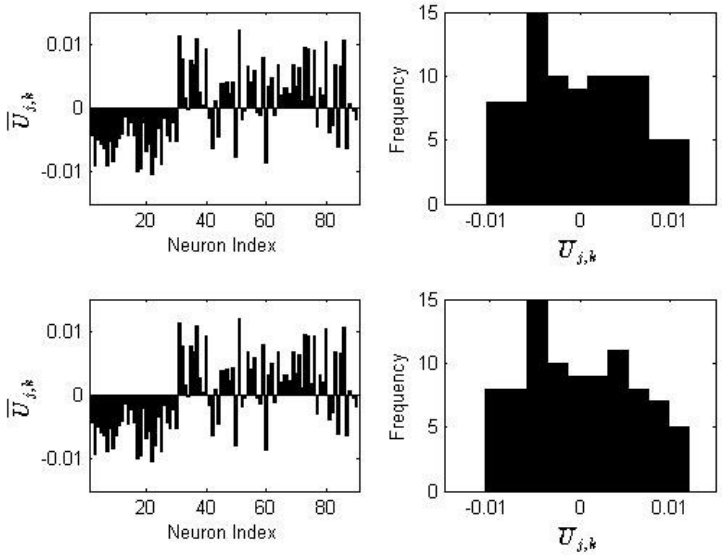

Figure 4.5: Histogram of VTS results using the watch interrupt model. Much more variance when compared to the Java simulation results in Figure 4.4.

4.4.2 Java and VTS Large Change Interrupt Model

We also compare the large change interrupt model with the Java simulation which remains the same as in the previous section. In Figure 4.6 the heart amplitude and period

after the ischemia at t = 500 s differ respectively by approximately 20%. These differences exceed those observed between the VTS watch model and Java simulations by an order of magnitude. This observation is further exemplified by the neuron updating results presented in Figures 4.7 and 4.8.

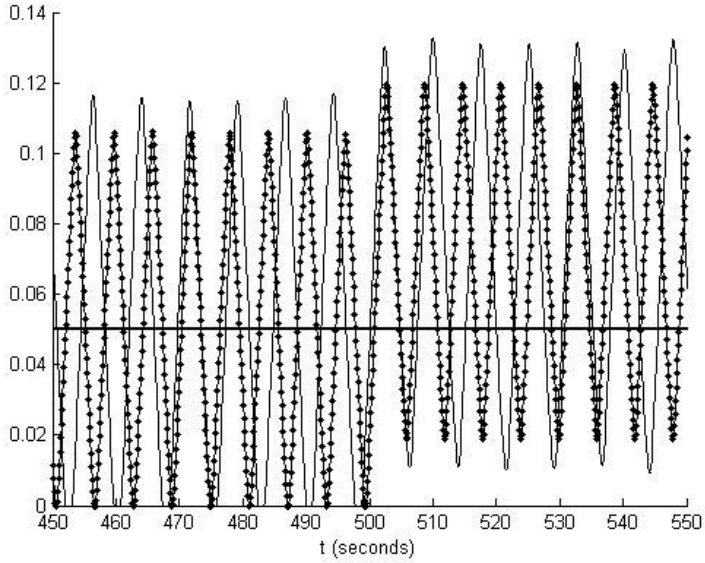

Figure 4.6: Scaled heart rate (dimensionless, on the y-axis) plotted against simulation time for both Java and VTS. VTS heart rate is shown in black dotted lines and Java heart rate is shown in grey lines. Here the VTS simulation uses the large change interrupt model.

The differences between the watch and large change are particularly evident in the differences shown by the histograms of each model in Figures 4.5 and 4.8. The watch model also shows less neural updating at the cardiac level than the middle and top levels while the same is not true for the large change model.

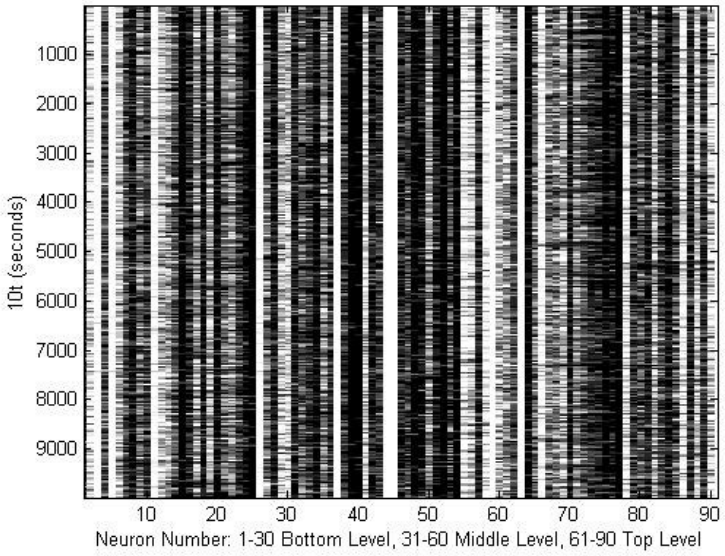

Figure 4.7: Neural network in VTS for raw updates per time step but using the large change interrupt model. Compared to Figure 4.3, the updates between the three layers are considerably more uniform.

The large change interrupt model appears to lead to a smaller bias in updating rates between levels than what we observe with the watch interrupt model. At the same time, the large change interrupt model favours the updating of a smaller set of neurons. As a result, there is less coherence in the response to external demand and, therefore, greater in heart rate than that observed for the watch interrupt model.

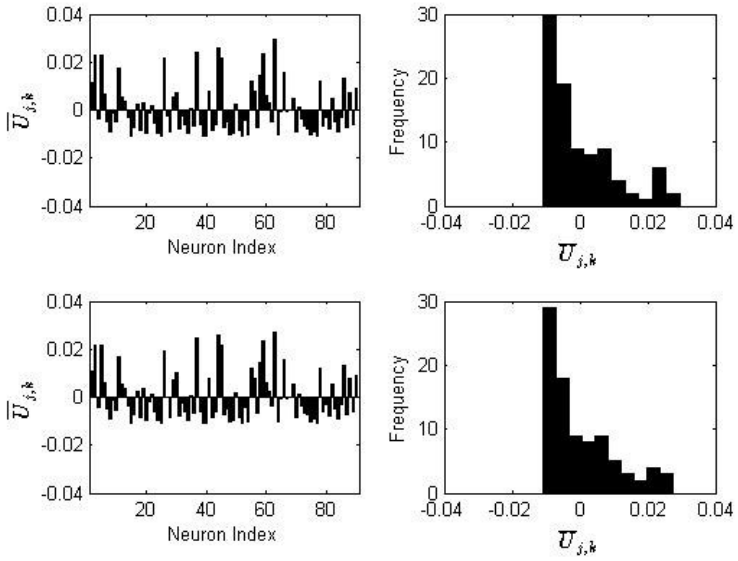

Figure 4.8: Histogram of VTS results using the large change interrupt model. Compared to the variation found in the VTS watch interrupt model, the variation from the large change model is right-sided due to an increase in positive neural activity updates. This is to be compared with the equally distributed minus and positive updates seen in Figure 4.5.

4.4.3 Summary

A comparison between three simulations involving the same network structure was made. The three simulations utilize sequential programming (Java), and a pair of event-driven simulations where neuron states are interrupted based on a watch interrupt model and a large change interrupt model. The results show that networks with the same structure can exhibit significantly different dynamics in the controlled variable—like heart rate. These simulations effectively involve *identical twins* in the sense that their network synaptic structure is identical. Thus, even identical twins that show nearly the same external traits may be undergoing quite different internal processing. The processing differences observed here between *twin networks* are further examined in the following sections in terms of network plasticity where neighbour weights may change and neural plasticity, which is associated with pathology.

4.5 NETWORK PLASTICITY ON, NEURAL PLASTICITY OFF

In this case we present results for both the constant neuron type and variable neuron type. First, we consider the constant neuron case with respect to observations made for the

previous case where network plasticity was off. Secondly, the variable neuron case is considered and compared to the constant neuron results for plasticity off and on. Maximal neighbour connectedness is used in the constant neuron type simulations while the variable neuron type is considered for both maximal and sparse neighbour connectedness.

4.5.1 Constant Neuron Type, Maximal Neighbor Connectedness

Heart rate derived from the Java model, VTS watch and VTS large interrupt models are respectively presented in Figures 4.9, 4.10 and 4.11. The heart rate is subjected to plasticity throughout the simulation period $0 \leq t \leq 1000$ and, as before, an ischemic event occurs over the range $500 \leq t \leq 1000$. Although the heart rate is bound within the same scale of 0–0.14, the results are different from those in the previous section where plasticity was turned off.

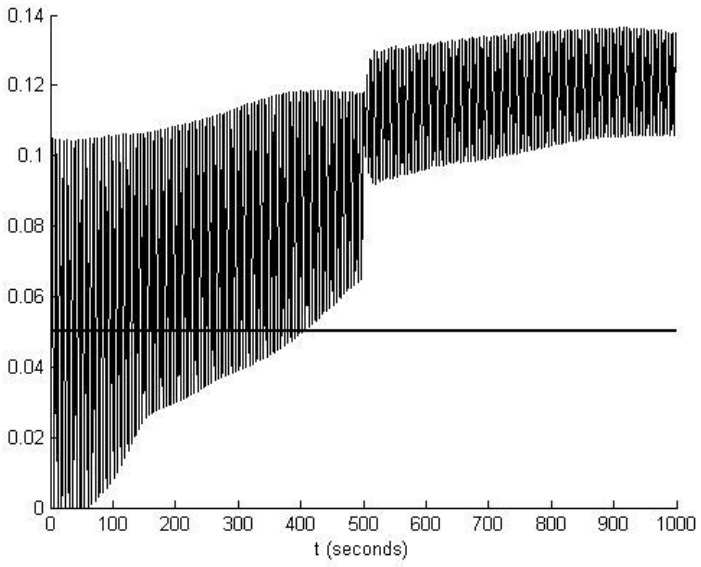

Figure 4.9: Network plasticity in the Java model. Scaled heart rate (dimensionless) is on the y-axis.

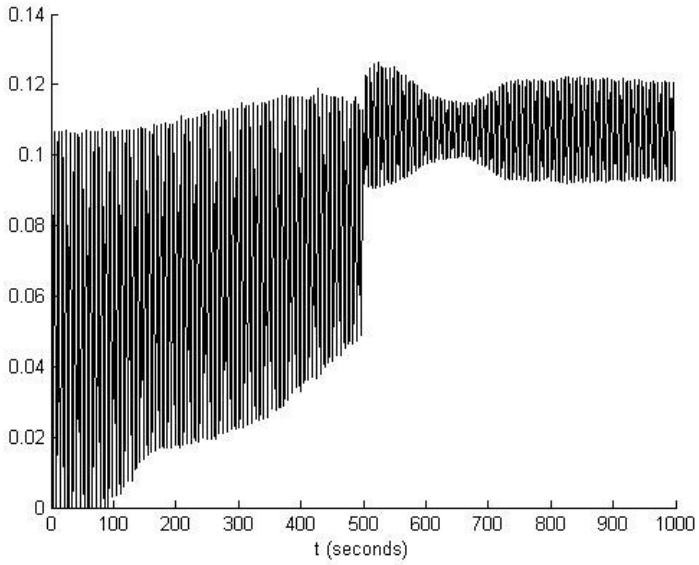

Figure 4.10: Network plasticity in the VTS watch interrupt change model. Ischemic event is applied at t = 500 seconds. Response is similar to that observed using the Java model; however, there is disturbance after the plasticity is applied. Scaled heart rate (dimensionless) is on the y-axis.

The raw updates per time step $U^{(n)}_{j,k}$ for each of the VTS watch and VTS large interrupt models are presented in Figures 4.12 and 4.13 and the results show a qualitative resemblance to their respective counterparts in Figures 4.3 and 4.7. Note that the Java simulation neuron updates must possess the same features as those in Figure 4.2.

If we continue and consider the average neuron updates $\overline{U}_{j,k}$ defined in Equation 4.1, size and frequency of updates is close for the respective models when plasticity is on or off as shown in the pair of Figures 4.5 and 4.14 and the large change model depicted in Figures 4.8 and 4.15.

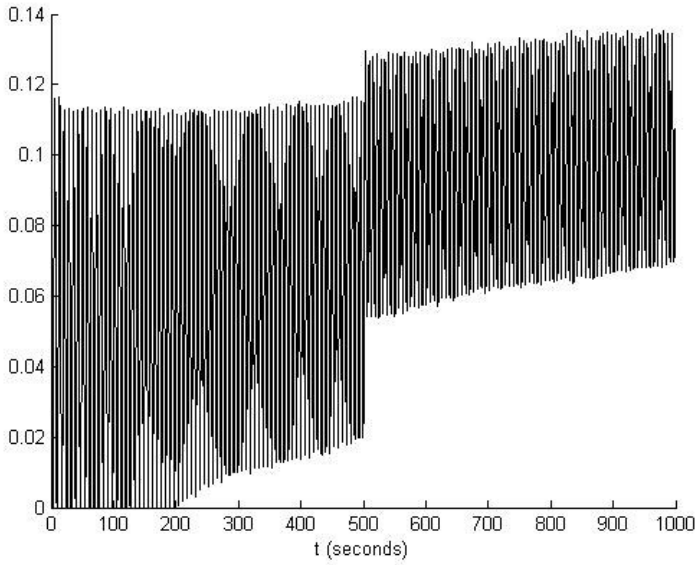

Figure 4.11: Network plasticity in the VTS large interrupt model. Ischemic event is applied at t = 500 seconds. Response is similar to the Java model but with more variation in heart rate after the plasticity is applied. Scaled heart rate (dimensionless) is on the y-axis.

Finally, we consider the extent to which the network is re-wired through changes in synaptic plasticity driven by the Hebbian and homeostatic rule base. In Figures 4.16, 4.17 and 4.18 the average neighbour weight is presented for the Java simulation, VTS watch and VTS large interrupt models, respectively. The average neighbour importance within each layer is

$$\overline{P}_j^{(n)} = \frac{\sum_{k=1}^{N_j} \sum_{i=1}^{N_{b(j,k)}} P_{J(i),K(i)}^{(n)}}{N_j} \tag{4.2}$$

and the network average is

$$\overline{P}(n) = \frac{\sum_{j=1}^{3} N_j \overline{P}_j^{(n)}}{\sum_{j=1}^{3} N_j} \tag{4.3}$$

which depends upon the fractional contribution of each layer $(N_j/\Sigma^3_1 N_j)\,\overline{P}_j^{(n)}$, $j = 1, 2, 3$. In our simulations $N_j = N, j = 1, 2, 3$ and the fractional contribution is simply 1/3 of the average layer importance $\overline{P}_j^{(n)}$. In Figures 4.16, 4.17 and 4.18 the fractional contribution of each layer to the network neighbour importance is given.

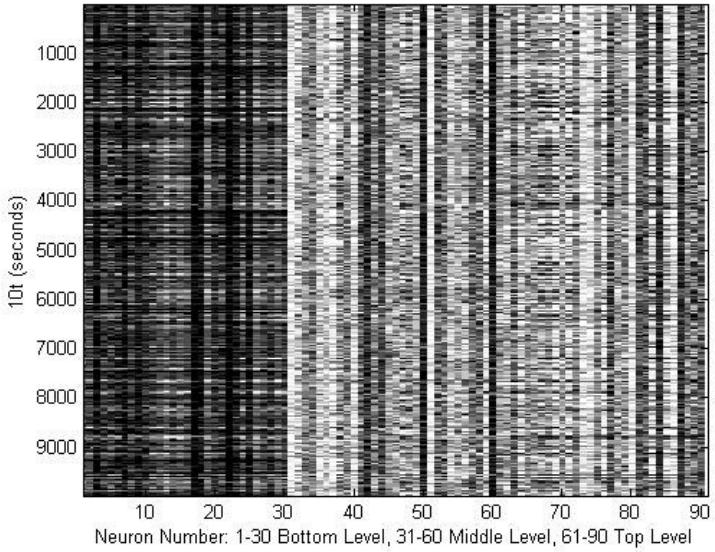

Figure 4.12: Neural network for VTS watch interrupt model and network plasticity applied. Similar results to Figure 4.3 with more activity in the bottom later compared to the other two layers.

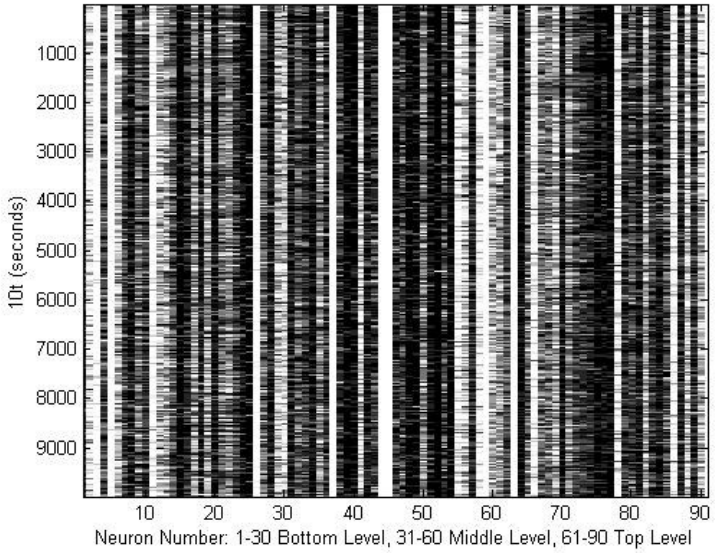

Figure 4.13: Neural network for VTS large interrupt model and network plasticity applied. There is roughly equal activity amongst the three neural layers.

All simulations begin with an average neighbour weight that is equal for each of the three layers, *i.e.* $\overline{P}_j^{(n)} = 1/3, j = 1, 2, 3$. It is important to note that the neurons have equal access to neighbours on average but the layers do *not* since neurons within a layer may only connect to an adjacent layer. If each layer was to have equal overall weight within the

network with respect to the magnitude of its neighbour weights then the top and bottom layers would contribute 2/7 or 0.29 of the total with the middle contributing the remaining 3/7 or 0.43. This observation helps to explain why most simulations give rise to an increased fractional contribution of the middle layer beyond the initial 1/3 that it was awarded. However, the levels reached for the middle layer are significantly below 0.37 where equality between layers would be established. Note that the middle level does, in general, have increasing dominance in the presence of network plasticity giving a greater integration of information due to increased synaptic weight at that level.

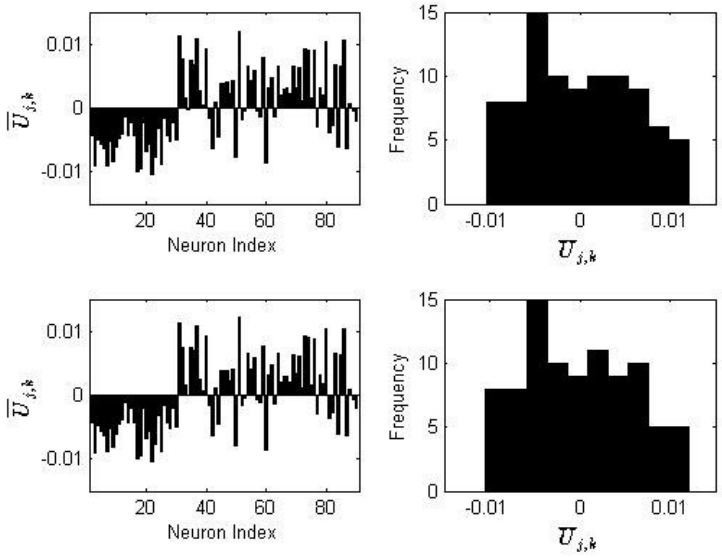

Figure 4.14: Histogram of VTS results using the watch interrupt model with network plasticity. Variation is nearly identical to the baseline VTS results.

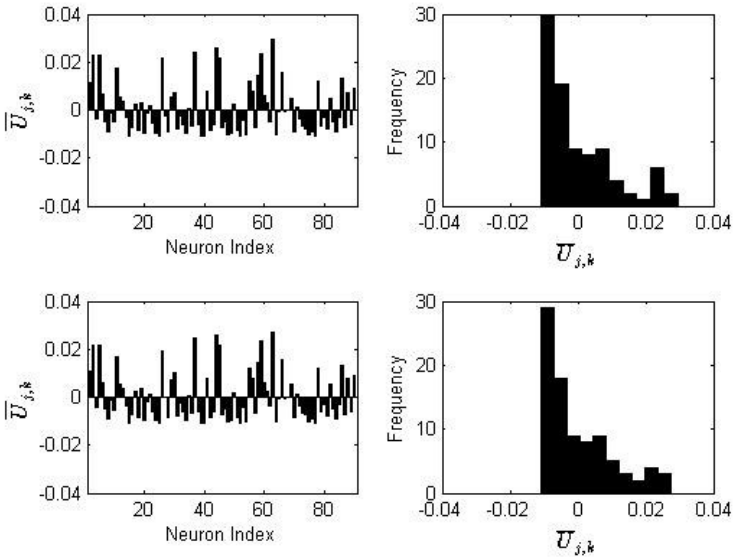

Figure 4.15: Histogram of VTS results using the large change interrupt model with network plasticity. Compared to the variation found in the VTS watch interrupt model, the variation from the large change model is right-sided compared to being equally distributed as in Figure 4.8.

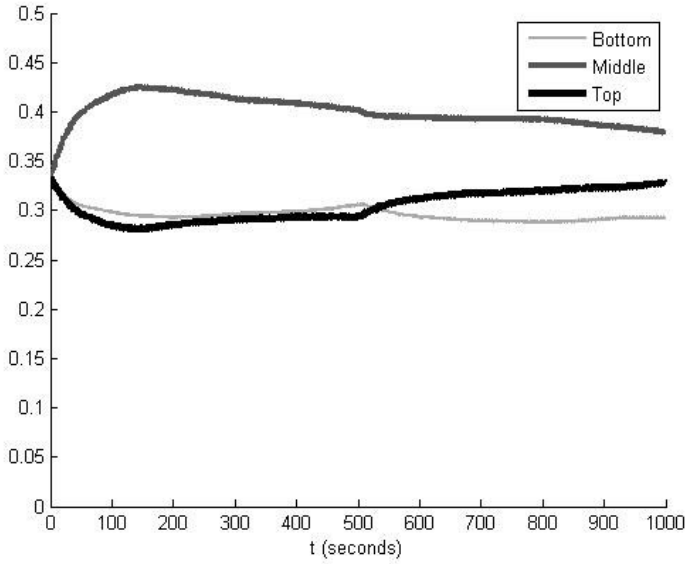

Figure 4.16: Average neighbor weights using the Java model with network plasticity. Average neighbor weight is on the y-axis.

The primary difference between the simulation approaches lies with the distribution of average weights between the layers. In the sequential implementation the middle layer has increased synaptic weights while the top and bottom are both reduced. On the other hand, the VTS watch simulation leads to a wider spread between the top and middle weights, while the VTS large change interrupt model leads to the smallest variation.

Finally, all simulations generally tend to increase the middle layer neighbour importance and simultaneously decrease the top and bottom layer neighbour importance.

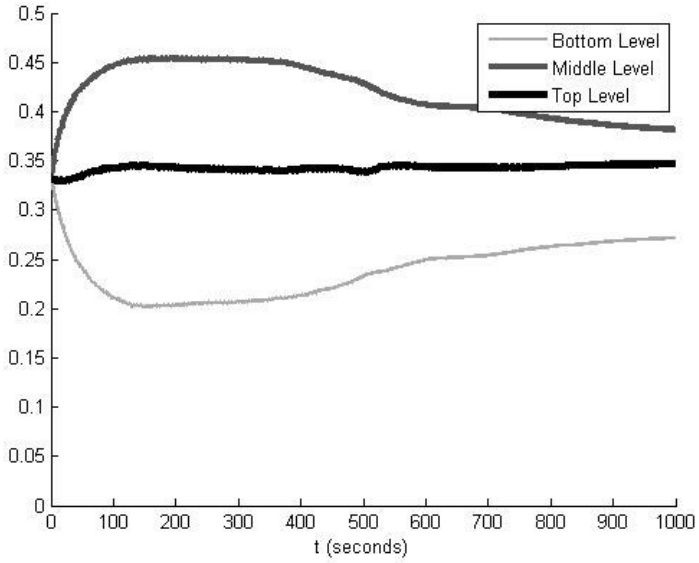

Figure 4.17: Average neighbor weights using the VTS watch interrupt model with network plasticity. Average neighbor weight is on the y-axis. Compared to the Java results, the amplitude of the bottom layer is quite different. This is due to increased activity in the bottom layer using this simulation model.

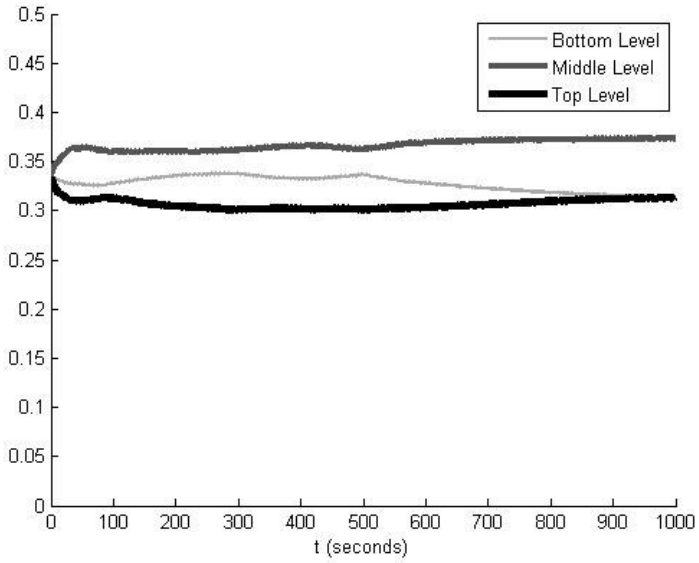

Figure 4.18: Average neighbor weights using the VTS large interrupt model with network plasticity. Average neighbor weight is on the y-axis. Amplitudes are much tighter compared to both Java and VTS watch interrupt models.

4.5.2 Variable Neuron Type, Maximal Neighbor Connectedness

We continue to consider networks with variable neuron type and again we consider heart rate derived from the Java model, VTS watch and VTS large interrupt models that are respectively presented in Figures 4.19, 4.20 and 4.21.

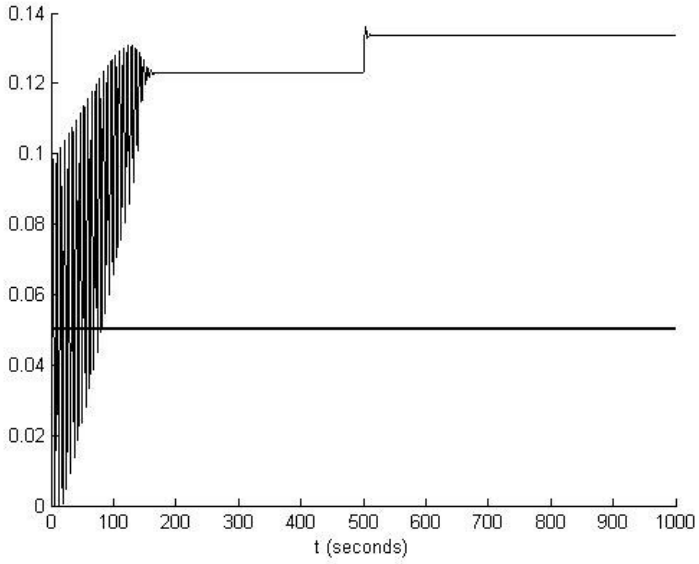

Figure 4.19: Variable neuron effects on heart rate using the Java simulation. Scaled heart rate (dimensionless) is on the y-axis. Note that there is essentially steady-state dynamics after 150 s (other than the ischemia applied at $t = 500$ s).

The variable neuron model is intended to represent the varied nature of neural properties within a network. In particular, neural types in cardiac control show a localized structure: the top layer has a greater sensitivity to external demand and might be termed *demand-centric*, while the bottom layer is more attuned to sensory feedback of heart rate and is analogously *heart-centric*. This localization of neural types represents the differing responsibilities of the top and bottom layers. We consider the extent to which these differing neuron types change how network plasticity evolves.

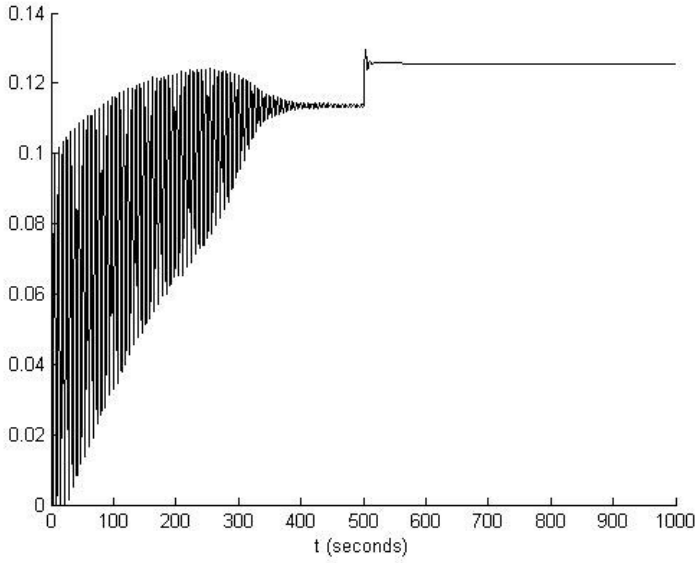

Figure 4.20: Variable neuron effects on the heart rate using the VTS watch interrupt model. Scaled heart rate (dimensionless) is on the y-axis. Compared to the Java model, the VTS watch interrupt model takes longer to reach steady state (300 s for VTS versus about 150 s for Java).

Starting with the Java simulation in Figure 4.19 and comparing with the VTS watch (Figure 4.20) and VTS large change model (Figure 4.21) it is clear that all simulations proceed to a reasonable approximation of the same steady-state heart rate. However, the dynamics in the progress toward the steady-state is different for each simulation. The sequential simulation shows the most rapid convergence of heart rate dynamics while the VTS watch and VTS large interrupt models show a slower convergence and are quite similar.

A further comparison of the variable neuron type results in Figures 4.19, 4.20 and 4.21 with their constant neuron type counterparts presented in Figures 4.9, 4.10 and 4.11 shows a dramatic shift in the steady-state behaviour. The variable neuron type simulation shows an extinction of the oscillatory behaviour followed by nearly constant behaviour which does not occur in the constant neuron type results.

The raw updates per time step $U^{(n)}_{j,k}$ for each of the VTS watch interrupt and VTS large change interrupt models are presented in Figures 4.22 and 4.23. The results show a qualitative resemblance to their respective counterparts in Figures 4.3 and 4.7. Note that

the Java simulation neuron updates must possess the same properties as those shown in Figure 4.2, and are therefore not shown.

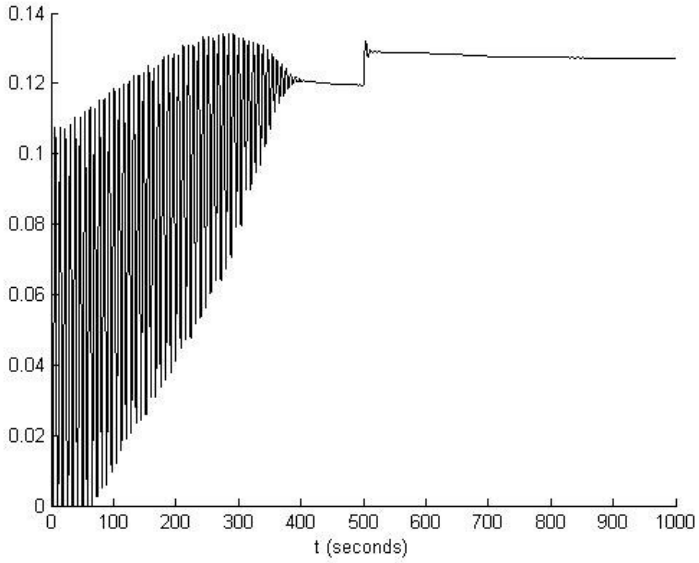

Figure 4.21: Variable neuron effects on the heart rate using the VTS large interrupt model. Scaled heart rate (dimensionless) is on the y-axis. Compared to the other VTS model, the results are very similar.

We consider the average neuron updates $\overline{U}_{j,k}$, as defined in Equation 4.2. The size and frequency of updates is in relative agreement for the respective models when plasticity is on or off.

Finally, we consider the extent to which the network is *re-wired* through changes in synaptic plasticity driven by the Hebbian and homeostatic rule base. In Figures 4.26, 4.27 and 4.28 the average neighbour weight is presented for the Java simulation and the VTS watch and VTS large interrupt models, respectively. All of the models commence the simulation with an equal average neighbour weight for each of the three layers. In all cases, the middle layer ends up with an increased dominance in neural activity, giving a greater integration of information due to increased synaptic weight at that level (intra-thoracic).

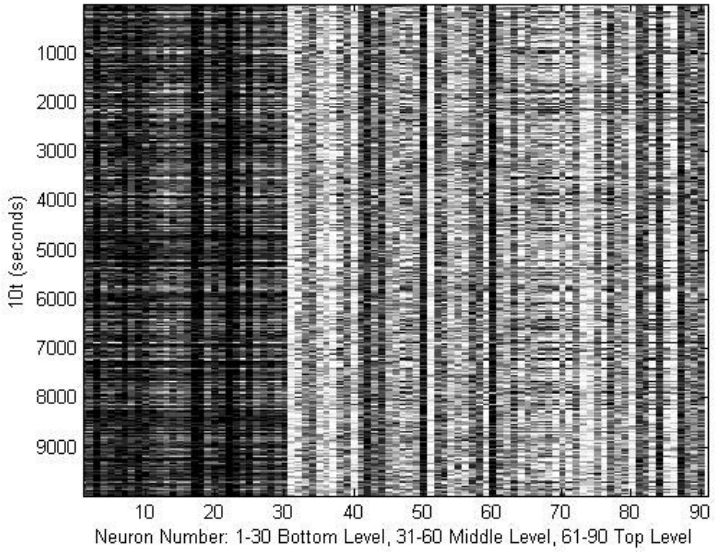

Figure 4.22: Neural network for VTS watch interrupt model using variable neuron effects. There is a more activity noted in the bottom layer.

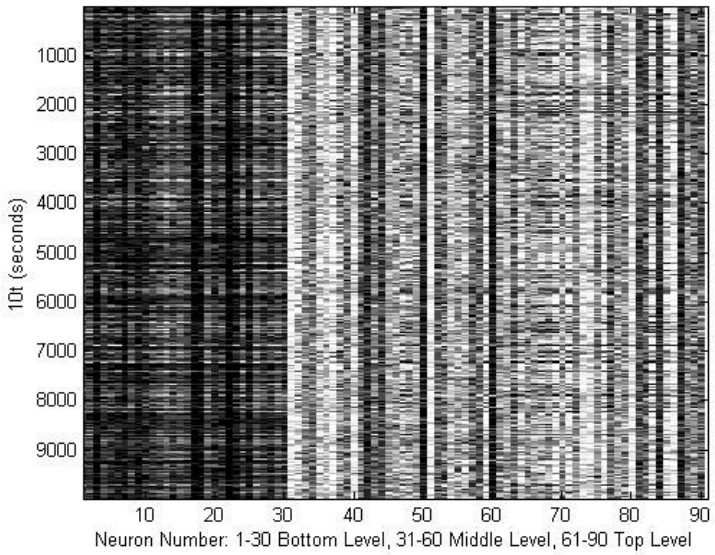

Figure 4.23: Neural network for VTS large interrupt model using variable neuron effects. There is a more activity noted in the bottom layer equal to the VTS watch interrupt model.

The primary difference between the simulation methods with variable neuron type relates to the distribution of weights between the layers. In the sequential implementation, the middle layer has increased synaptic weights, while the top and bottom are typically reduced. On the other hand, the VTS watch simulation leads to a wider spread (as

indicated by the histograms in the figures) between the top and middle weights while the VTS large change interrupt model leads to the smallest variation.

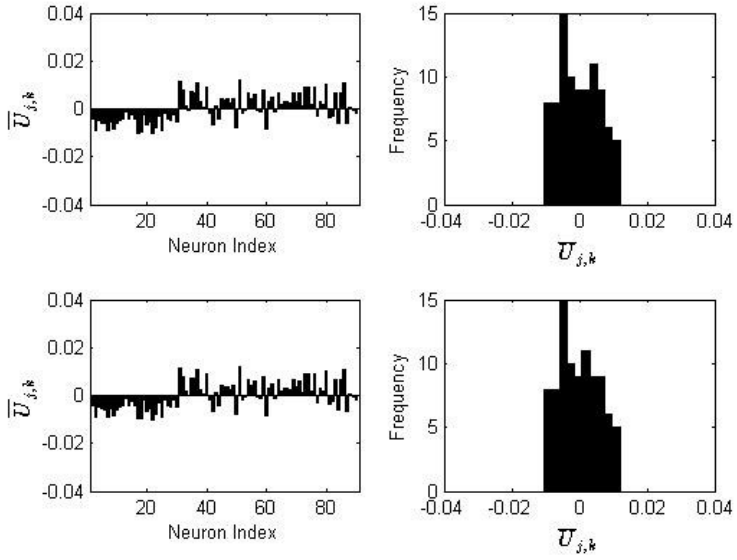

Figure 4.24: Histogram of VTS results using the watch interrupt model with variable neuron effects. Variation here is slightly less compared to previous simulations and equally distributed around zero.

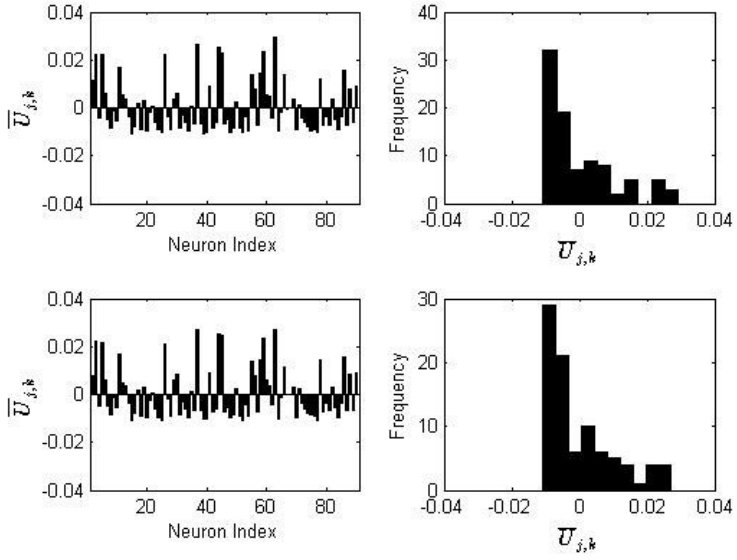

Figure 4.25: Histogram of VTS results using the large interrupt model with variable neuron effects. Variation here is slightly less compared to previous simulations, but here the distribution is right-sided around about -0.01.

If we compare these variable neuron type results with those for the constant neuron type, we observe a reduction in the top layer importance for the variable neuron type as

opposed to the bottom layer importance which is reduced for the variable neuron type. Thus, the variable neuron type with network plasticity has a tendency to shift the overall importance toward the middle layer but also to reduce the importance of the uppermost layer which is *demand-centric* in favour of the lower layer which is *heart-centric*.

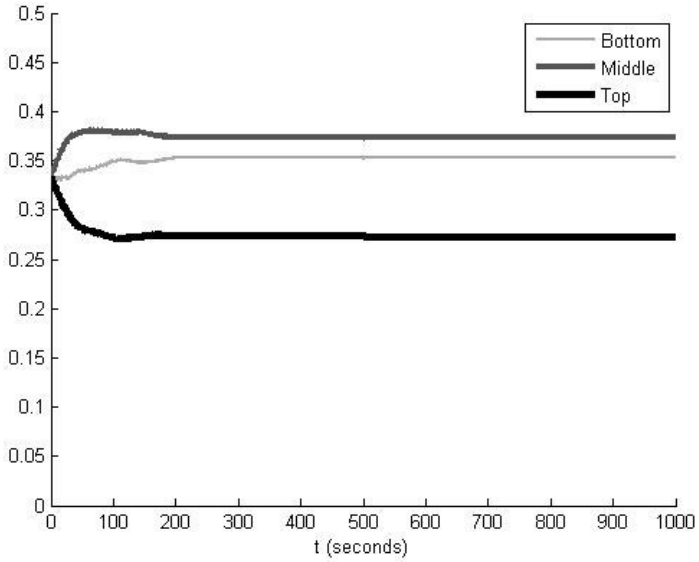

Figure 4.26: Average neighbor weights using the Java model with variable neuron effects. Average neighbor weight is on the y-axis.

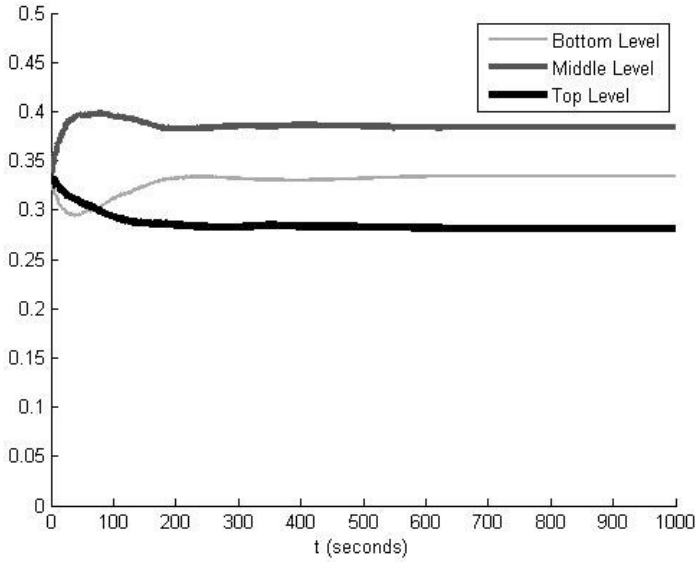

Figure 4.27: Average neighbor weights using the VTS watch interrupt model with variable neuron effects. Average neighbor weight is on the y-axis. Compared to the Java results, the bottom layer's shows an early dynamic not seen in the previous figure.

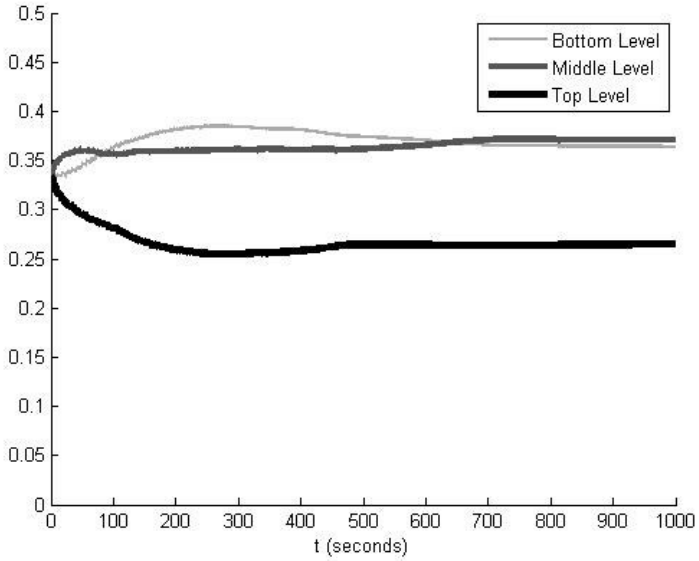

Figure 4.28: Average neighbor weights using the VTS large interrupt model with variable neuron effects. Average neighbor weight is on the y-axis. Compared to the VTS watch interrupt model, the bottom layer again shows different dynamics.

4.5.3 Variable Neuron Type, Sparse Neighbor Connectedness

In all of the simulations considered so far we have assumed a maximal neighbour connectedness. This was justified when neural and network plasticity were both off; however, when either or both of network and neural plasticity are on there is an increased sensitivity to neighbour connectedness that is not seen in the absence of plasticity. In particular, we find sparsely connected networks are less responsive to external stimuli relative to networks with maximal connectedness. These observations may be relevant to the nature of aging and its effect on cardiac control.

The results for sparse neighbour connectedness are respectively presented for the Java model, VTS watch and VTS large interrupt models in Figures 4.29, 4.30 and 4.31 and are compared with their counterparts in Figures 4.19, 4.20 and 4.21.

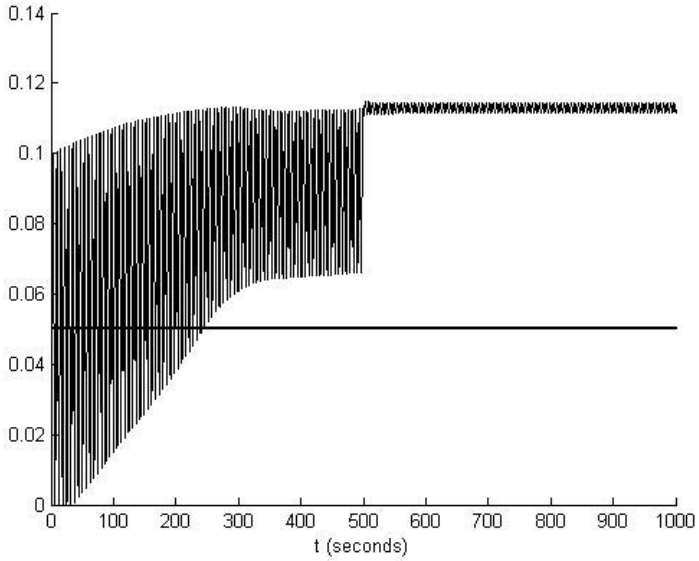

Figure 4.29: Variable neuron effects on heart rate using the Java simulation with sparse neighbor connectedness. Scaled heart rate (dimensionless) is on the y-axis. This is similar to the Java simulation without sparse neighbor connectedness but the amplitude of oscillation after the event at $t = 500$ seconds is suppressed significantly (quick damping).

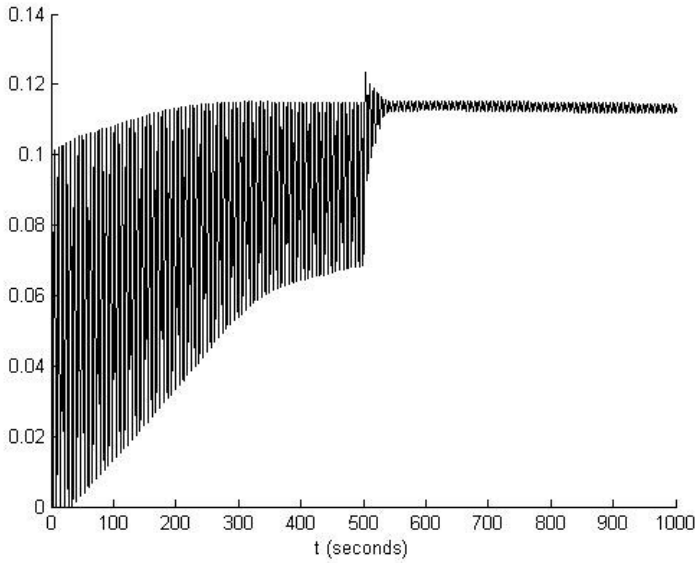

Figure 4.30: Variable neuron effects on heart rate using the VTS watch interrupt model with sparse neighbor connectedness. Scaled heart rate (dimensionless) is on the y-axis. Result is similar to the Java model.

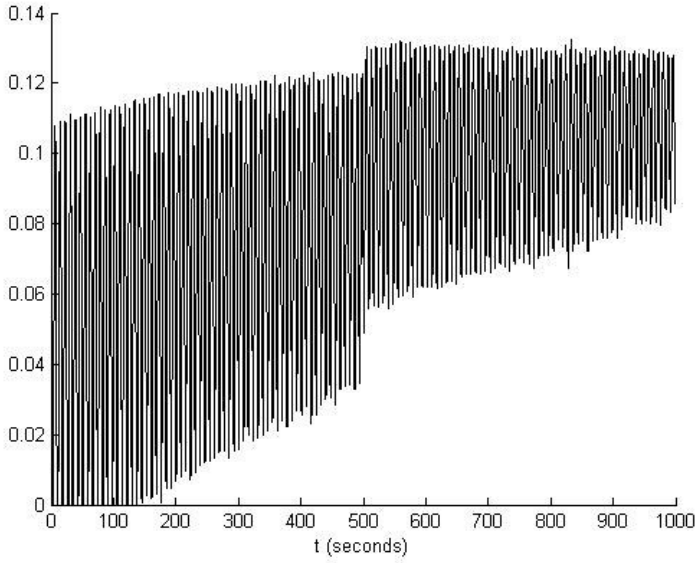

Figure 4.31: Variable neuron effects on heart rate using the VTS large interrupt model with sparse neighbor connectedness. Scaled heart rate (dimensionless) is on the y-axis. Result has much more variance after $t = 500$ seconds compared with the Java and VTS watch interrupt models. Although the simulation was not run longer, it is likely that the oscillations will eventually die out as was observed in previous figures.

In all of the simulations the oscillatory behaviour in heart rate is not extinguished for the sparsely connected networks as it is for the maximally connected simulations. Most surprisingly, the heart rate in the sequential (Figure 4.29) and the VTS watch (Figure 4.30) models are qualitatively similar, both in the evolution toward the steady-state and in the steady-state reached (they would become equal in the limit of a vanishing time step). In complete contrast, the large change event-driven results for the maximally connected network (Figure 4.21) and sparsely connected network (Figure 4.31) are quite different.

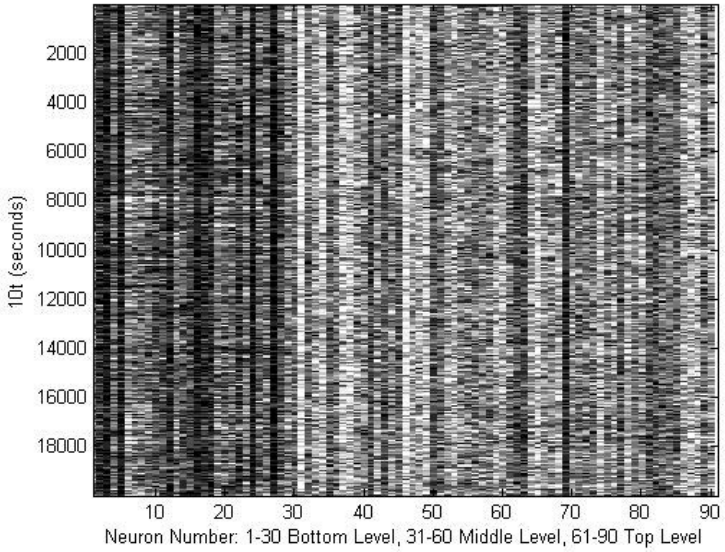

Figure 4.32: Neural network for VTS watch interrupt model using variable neuron effects and sparse neighborhood connectedness. There is a slightly more activity noted in the bottom layer but not as distinct compared with previous VTS simulations.

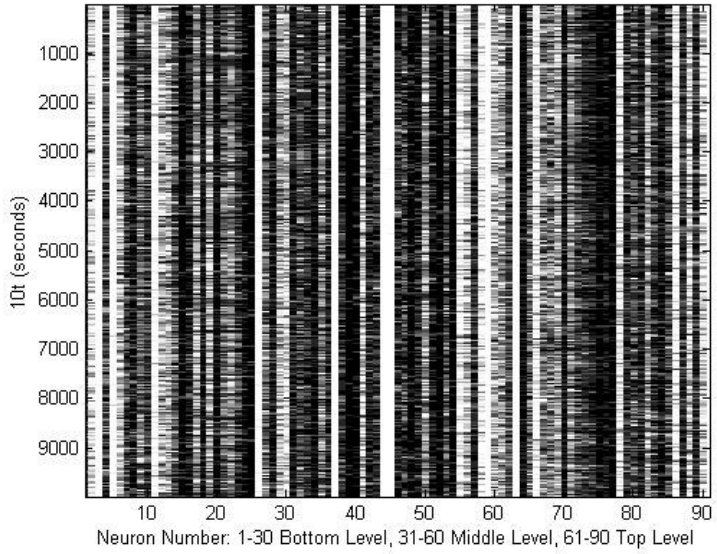

Figure 4.33: Neural network for VTS large change model using variable neuron effects and sparse neighborhood connectedness. There is roughly equal activity amongst the layers in this model.

We present, as before, the results that show a close similarity in the nature of neural processing between networks with maximal and sparse neighbour connectedness. In particular, the raw updates per time step $U^{(n)}_{j,k}$ for each of the VTS watch and VTS large interrupt models are presented in Figures 4.32 and Figure 4.33 and the results show a

qualitative resemblance to their respective counterparts in Figures 4.22 and 4.23. Note that the Java simulation neuron updates must possess the same features as those in Figure 4.2 since the neuron updating is chosen independently of the network structure or neural states. It is interesting that the similarity between heart rate in the sequential and VTS watch simulations is not replicated in the network processing as one might have expected.

A similar point is observed in the average neuron updates $\overline{U}_{j,k}$ defined in Equation 4.2. The size and frequency of updates is similar for the respective models with plasticity on for both the sparse and maximal connectedness respectively in figure pairings: (i) watch interrupt model (Figures 4.34 and 4.24) and (ii) large interrupt model (Figures 4.35 and 4.36).

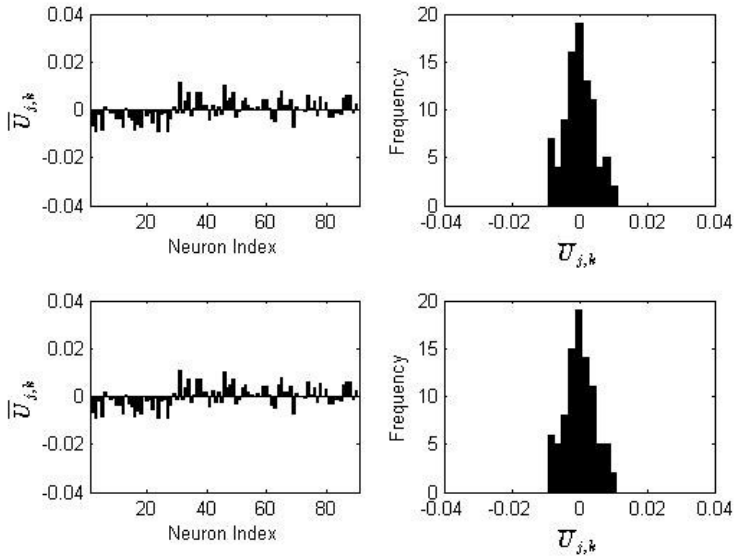

Figure 4.34: Histogram of VTS results using the watch interrupt model with variable neuron effects and sparse neighbor connectedness. Variation here is equally distributed around zero.

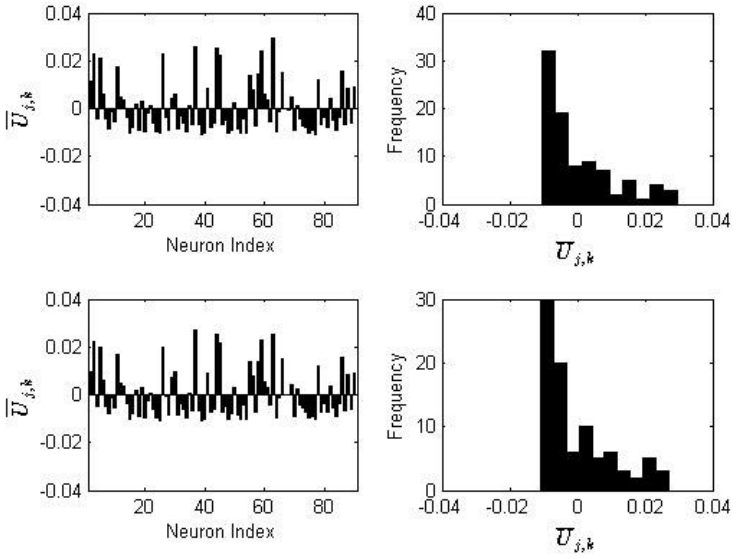

Figure 4.35: Histogram of VTS results using the large interrupt model with variable neuron effects and sparse neighbor connectedness. Variation here is distributed in a right-sided manner.

Network re-wiring due to synaptic plasticity is shown in Figures 4.36, 4.37 and 4.38. If each of these figures are respectively compared to their maximally connected counterparts in Figures 4.26, 4.27 and 4.28, we see that they are quite similar.

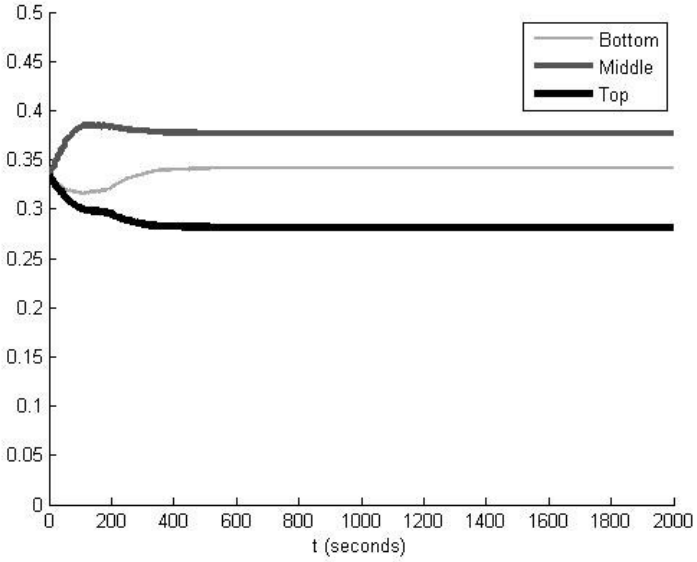

Figure 4.36: Average neighbor weights using the Java model with variable neuron effects and sparse neighbor connectedness. Average neighbor weight is on the y-axis.

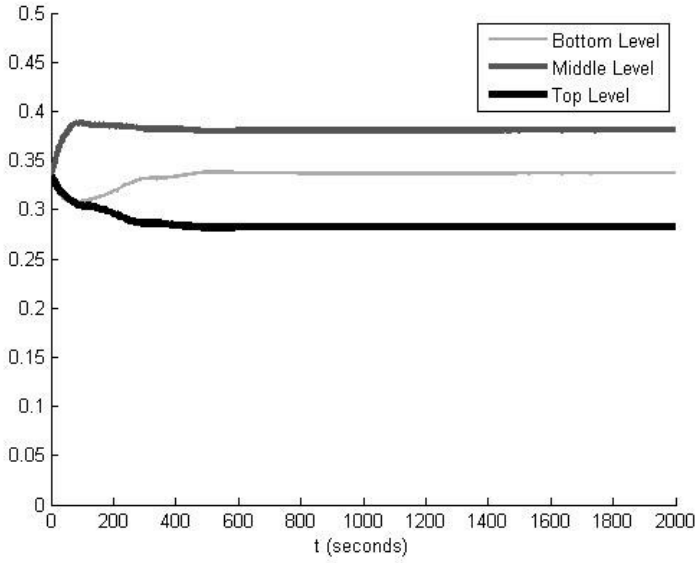

Figure 4.37: Average neighbor weights using the VTS watch interrupt model with variable neuron effects and sparse neighbor connectedness. Average neighbor weight is on the y-axis. Results are very similar to those obtained using the Java simulation.

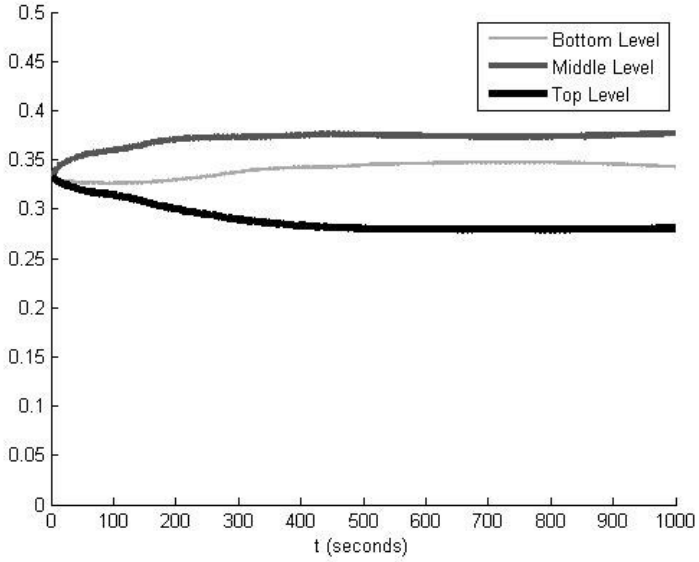

Figure 4.38: Average neighbor weights using the VTS large interrupt model with variable neuron effects and sparse neighbor connectedness. Average neighbor weight is on the y-axis. Results are similar to both the Java and VTS watch interrupt simulation methods.

4.5.4 Summary

Two observations may be made with respect to plasticity for constant neuron type: (i) differences in network processing between simulations where plasticity is turned off or

on are not significantly altered in spite of altered synaptic weights within the network when plasticity is turned on, and (ii) altering of the network synaptic weights due to plasticity does, however, lead to significant alterations in heart rate dynamics in spite of the network processing remaining qualitatively the same. This pair of observations point once again to the property that the relationship between a network's organizational structure and the controlled variable (heart rate) is so complex, even for the small network studied here, that the controlled variable is a difficult variable to use as a predictor of network properties. While this point may seem obvious from the context of this mathematical model, clinical practice does tend to emphasize the importance of indices such as heart rate, blood pressure, etc. and their relation to the nervous system. Although such a relationship based on plasticity may seem intuitively correct it is based upon assumptions about the nature of nervous control that do not include neural networking of cardiac control.

Two observations may be made with respect to neighbour connectedness and its impact on network plasticity: (i) the nature of updating has a significant impact on heart rate dynamics in the controlled variable with the sequential and VTS watch models yielding qualitatively similar results and (ii) all simulations show a longer time is required to reach a steady-state behaviour before the onset of the myocardial ischemia is applied ($t = 500$ s). It appears that reduced neighbour connectedness also reduces the number of neural updates within the network. Hence, the entire network has a reduced sensitivity to sensory feedback of heart rate and this is observed in the tendency of the controlled variable (heart rate) to take longer to reach a steady-state. These effects are further examined in the next section, in light of autonomic derangement which we call *neural plasticity*.

4.6 Network Plasticity On, Neural Plasticity On

In this case, both network plasticity (last section) and neural plasticity (this section) are turned on. The results are most easily presented separately for each of the simulation types and we start with the sequential Java simulation followed by the event-driven VTS watch and VTS large change interrupt simulation models. We will confine ourselves to

the consideration of constant neural type network since the thrust of our observations may be made without consideration of the variable neural type networks, which show a qualitatively similar behaviour.

Neural plasticity is on when $t \geq 500$ seconds in each simulation and thus network plasticity is solely active when $t < 500$ seconds. During autonomic derangement (details in Section 4.2.2) the sensitivity of the network neurons to heart rate feedback $h^{(n)}_{j,k}$ is steadily reduced and their sensitivity to demand $d^{(n)}_{j,k}$ is steadily increased. The heart rate response during the autonomic derangement is most pronounced during $500 \leq t \leq 800$ since this is the domain within which the heart rate and demand sensitivities are changing. In other words, the heart rate and demand sensitivities reach their respective minima and maxima at approximately $t = 800$ seconds and note that this occurs because the average heart rate and demand sensitivities are in reasonable agreement. After the autonomic derangement has completed at t ~ 800 seconds, the network continues to evolve under the influence of network plasticity. Hence, each simulation has three parts: (i) region 1 where only network plasticity is active over $0 \leq t < 500$, (ii) region 2 where network plasticity and neural plasticity are active over the range $t = 500$ to $t \sim 800$, and (iii) region 3, approximately $t > 800$, wherein only network plasticity is active. The response within region 1 was considered in the last section but we enact it again here so that the neural plasticity is applied after network plasticity has had some time to exert an effect on the network synaptic strengths. The main interest here and the discussion surrounding the dynamics of the network response after neural plasticity is imposed at $t = 500$ and, specifically, during regions 1 and 2. Note that the boundaries of regions 2 and 3 are dependent upon the simulation and these will be redefined for the event-driven simulations.

4.6.1 Sequential Simulation in Java

The maximally-connected network, as shown in Figure 4.39 and sparsely-connected network, as shown in Figure 4.40 show different heart rate responses. In particular, region 2 of the maximally-connected network is free of oscillatory behaviour and shows a relatively large deviation in heart rate that settles in region 3 to a finite-amplitude oscillation. On the other hand, the sparsely-connected network shows neither of these

features and instead undergoes bursts in oscillatory behaviour within regions 2 and 3 that eventually settle to a constant steady-state. The heart rate responses show a rapid response to this change in plasticity.

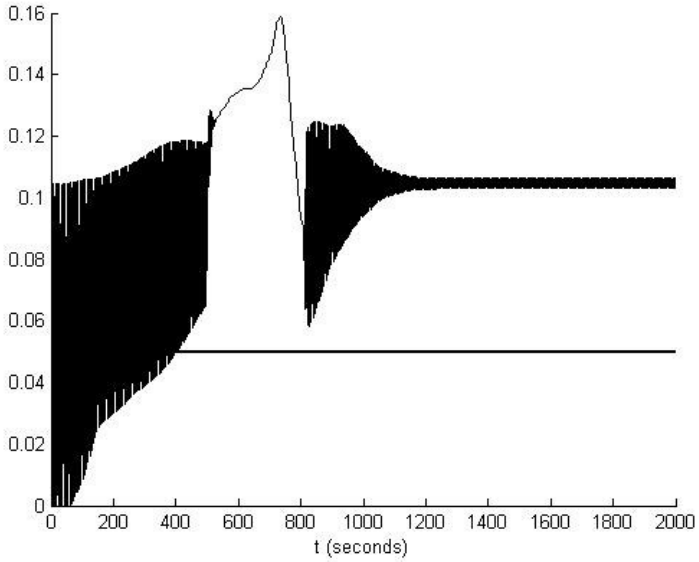

Figure 4.39: Effect on heart rate using the Java simulation with network and neural plasticity. This network is a maximally-connected network. Scaled heart rate (dimensionless) is on the y-axis. Network plasticity is active for $t < 500$ seconds. Maximum changes occur during $t = 500$ seconds and $t = 800$ seconds.

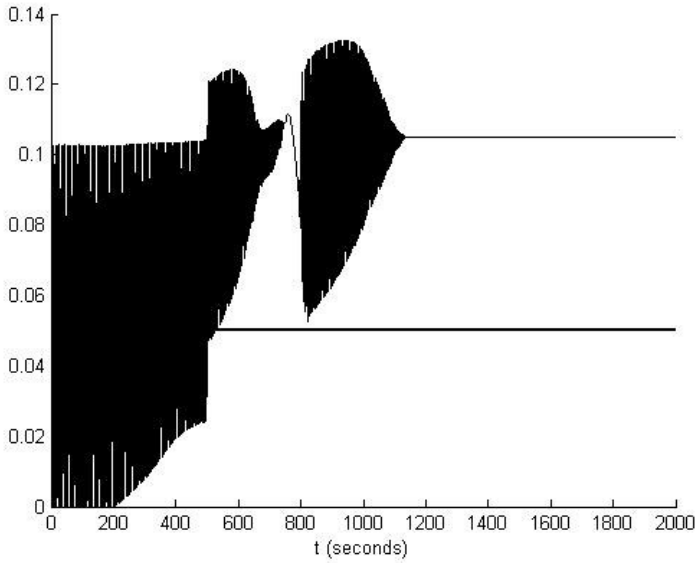

Figure 4.40: Effect on heart rate using the Java simulation with network and neural plasticity. This network is a sparse network. Scaled heart rate (dimensionless) is on the y-axis. Network plasticity is active for $t < 500$ seconds. Maximum changes occur during $t = 500$ seconds and $t = 800$ seconds. After $t = 800$ seconds, there is a significant period of oscillation before reaching steady state at $t = 1200$ seconds.

The evolution of the fractional contribution of each layer to the total network synaptic weights (shown in Figures 4.41 and 4.42) is rather similar in spite of the differences observed in the respective heart rate responses in Figures 4.39 and 4.40. Furthermore, it is clear that the autonomic derangement (region 2) has a strong impact on the evolution of network plasticity before (region 1) and after (region 3) the derangement has run its course. The fractional contribution for the maximally-connected network result with neural plasticity turned off is depicted in Figure 4.9 of the previous section and it is useful to compare that result with the same presented here in Figure 4.41. It is clear that when neural plasticity is turned off, the fractional contribution is settling out to a middle, bottom, and top arranged in decreasing magnitude.

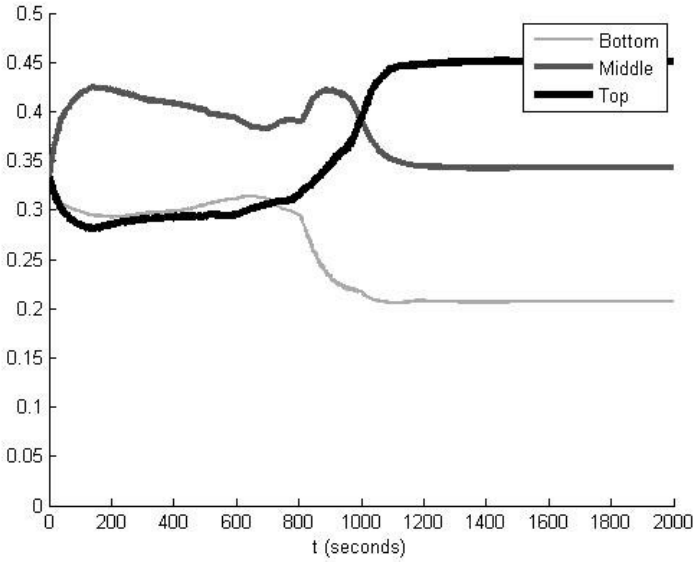

Figure 4.41: Average neighbor weights using the Java model with a maximally-connected network. Average neighbor weight is on the y-axis. Compared to previous simulations, there is much activity in all three layers before steady state is reached.

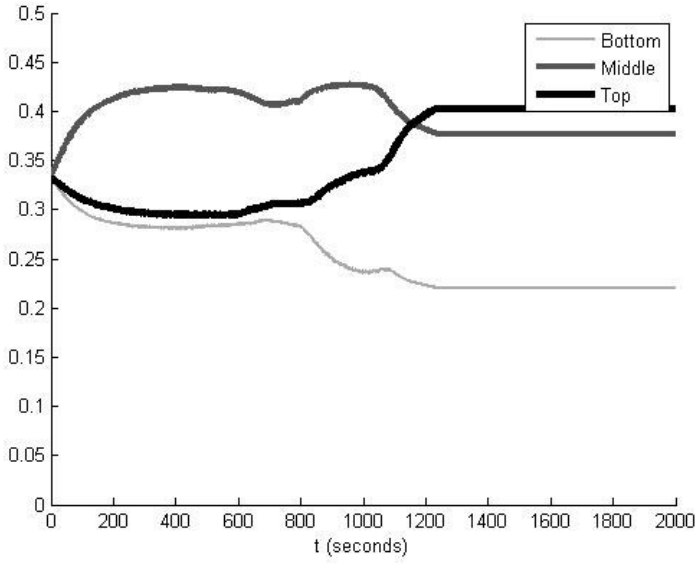

Figure 4.42: Average neighbor weights using the Java model with a sparse network. Average neighbor weight is on the y-axis. Compared to the maximally-connected network, the results are similar when considering dynamics.

Quite a different result is observed when neural plasticity is turned on, as shown in Figure 4.41. The middle, bottom and top undergo a dramatic shift to the decreasing arrangement, top, middle and bottom. This behaviour is seen in both of the maximally-connected and sparsely-connected network with neural plasticity turned on and is more marked in the former. The reduction in connectedness leads to less variability in the fractional contributions but quite different heart rate responses.

4.6.2 Event-driven Simulation: VTS Watch Interrupt Model

The maximally-connected network shown in Figure 4.43 and sparsely-connected network depicted in Figure 4.47 have similar heart rate responses and these simulations may be roughly divided into three regions: (i) region 1 where only network plasticity is active over $0 \leq t < 500$, (ii) region 2 where network plasticity and neural plasticity are active over the range $t = 500$ to $t \sim 1200$, and (iii) region 3, approximately $t > 1200$, wherein only network plasticity is active. Region 2, the zone of autonomic derangement (details Section 4.2.2), is approximately double that seen for the sequential simulation in the previous subsection. This occurs because the application of network and neural plasticity event-driven neural updates coincide with updates in neural activity level (as in the

sequential simulation), which leads to wide variation in the duration of autonomic derangement for some neurons compared with others.

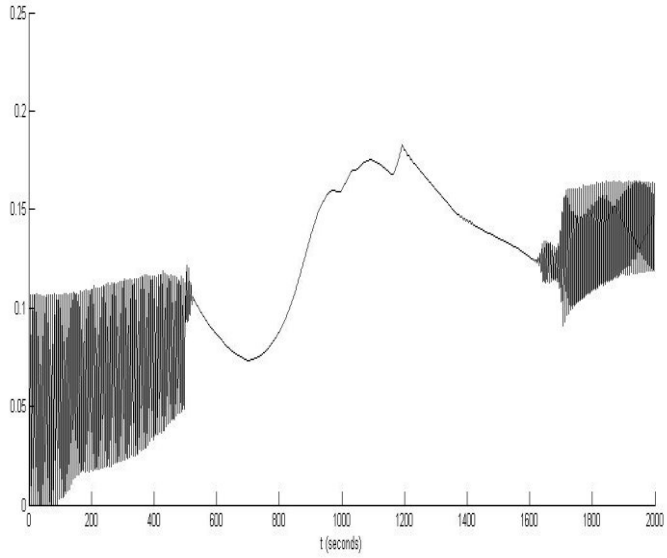

Figure 4.43: Effect on heart rate using the VTS watch interrupt model with network and neural plasticity. This network is a maximally-connected network. Scaled heart rate (dimensionless) is on the y-axis. Three distinct regions can be described: (i) network plasticity, (ii) network plasticity and neural plasticity and (iii) network plasticity. The autonomic derangement is observed in (ii) since neural plasticity involves the slow increase and decrease respectively of blood importance and heart importance.

It is interesting that, excluding the Java sequential sparsely-connected simulation in Figure 4.40, a comparison of heart rate responses in the event-driven results as shown in Figures 4.43 and 4.47 and sequential results depicted in Figure 4.39 indicate a similar shape for region 2 where only the width and height vary. The observed similarity is somewhat surprising because it exists in spite of the differences in the nature of processing within the network, *i.e.* neural and network plasticity are updated only when a neuron's activity level is updated.

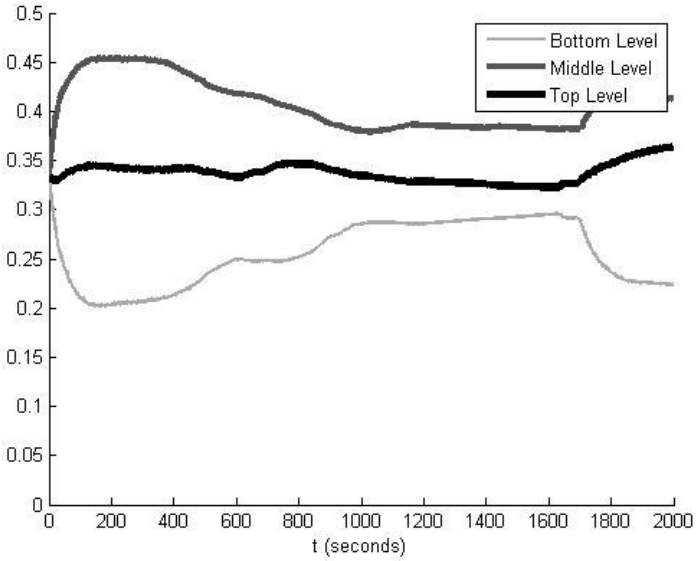

Figure 4.44: Average neighbor weights using the VTS watch interrupt model with network and neural plasticity. This network is a maximally-connected network. Average neighbor weight is on the y-axis.

The evolution of the fractional contribution to synaptic weights is also quite similar as shown for the maximally and sparsely-connected networks in Figures 4.44 and 4.48. These event-driven results differ from those shown for the sequential simulation since the even-driven simulation progresses toward a decreasing arrangement in magnitude of middle, bottom, and top that was typical of what was shown in the previous section (Section 4.5) where neural plasticity is turned off. This contrasts with the sequential simulation which tended toward a top, middle and bottom arrangement for decreasing magnitude in Figures 4.41 and 4.42.

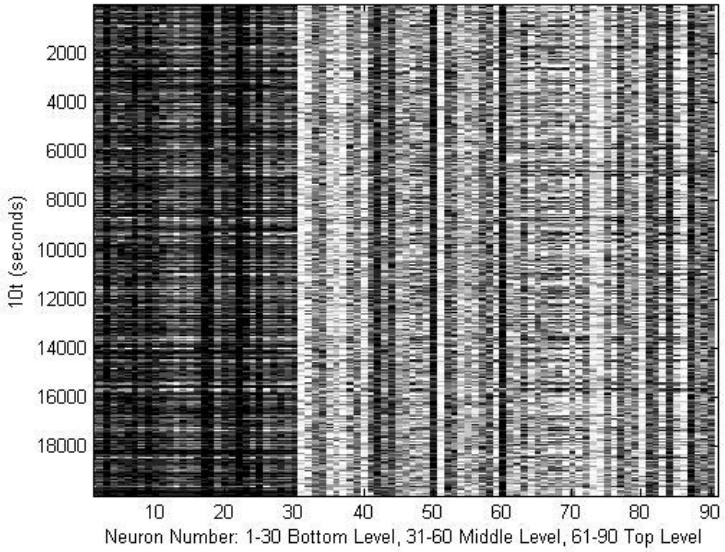

Figure 4.45: Neural network for VTS watch interrupt model using network and neural plasticity. This network is a maximally-connected network. There is slightly more activity noted in the bottom layer but not as seen with previous VTS simulations.

The nature of neural updating in the maximally-connected network considered in Section 4.5 where neural plasticity is turned off and presented in Figure 4.14 is quite similar to that seen here in Figure 4.50. This appears to indicate that neural plasticity does not have significant impact on the statistics of neural updating.

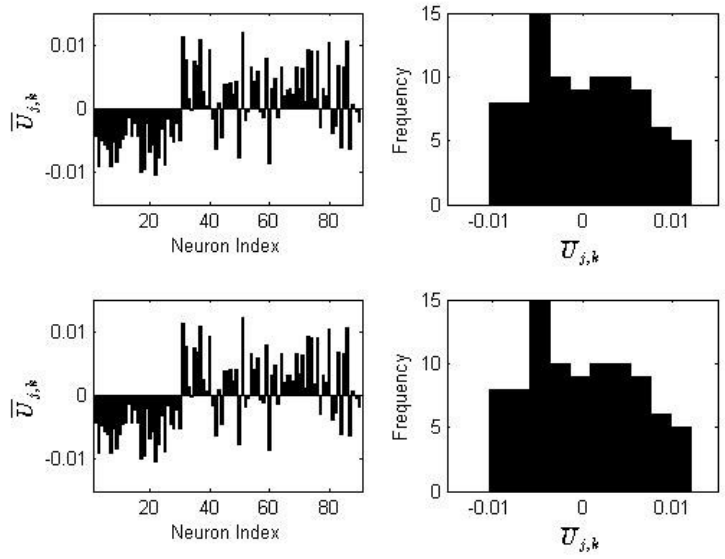

Figure 4.46: Histogram of VTS results using the watch interrupt model with network and neural plasticity with a maximally-connected network. Variation here is equally distributed around zero.

Finally, we note that the heart rate response seen for the sequential simulation maximally-connected network (Figure 4.39) and sparsely-connected network (Figure 4.40) are quite different and this entirely contrasts with the event-driven maximally-connected network shown in Figure 4.43 and the sparsely-connected network in Figure 4.47 that show similar heart rate responses.

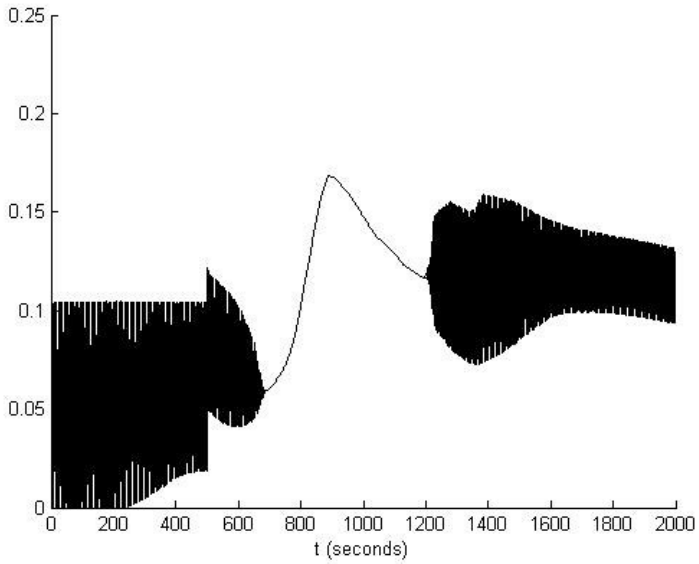

Figure 4.47: Effect on heart rate using the VTS watch interrupt model with network and neural plasticity. This network is a sparse network. Scaled heart rate (dimensionless) is on the y-axis. Three distinct regions can be described: (i) network plasticity, (ii) network plasticity and network plasticity and (iii) network plasticity. Results are similar to the maximally-connected network except region (ii) is shorter and reaches the maximum heart rate quicker.

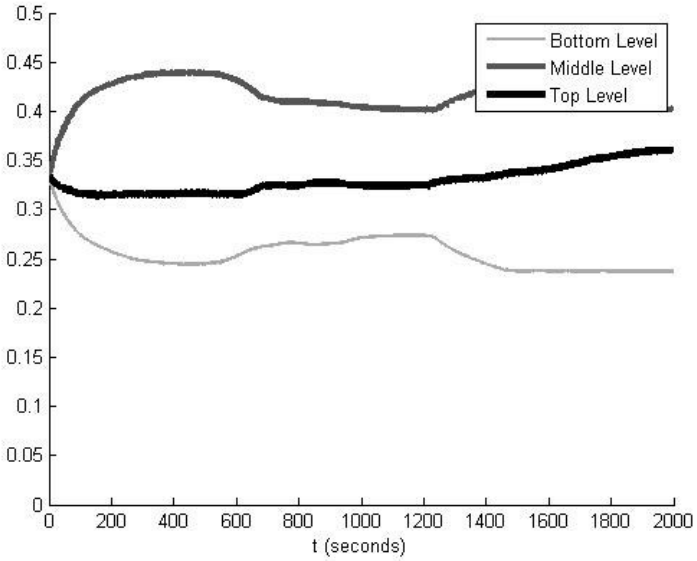

Figure 4.48: Average neighbor weights using the VTS watch interrupt model with network and neural plasticity. This network is a sparse network. Average neighbor weight is on the y-axis. This is a similar result to that shown for the maximally-connected network.

This implies that there is a level of robustness to parametric change that resides in the event-driven simulation. This type of observation is important from a physiological perspective since it implies that the mechanisms that drive the timing of changes in living systems, just like the delivery of a joke, impact the final results and, therefore, should be considered.

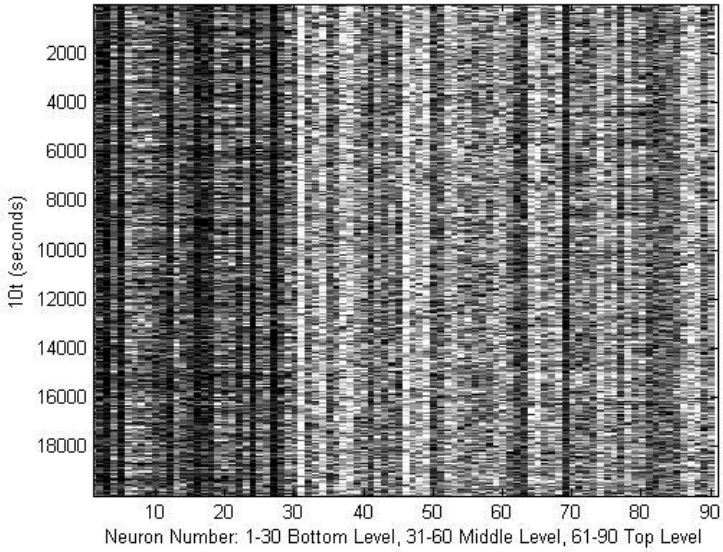

Figure 4.49: Neural network for VTS watch interrupt model using network and neural plasticity. This network is a sparse network. There appears to be more activity on the bottom layer of the network in this simulation.

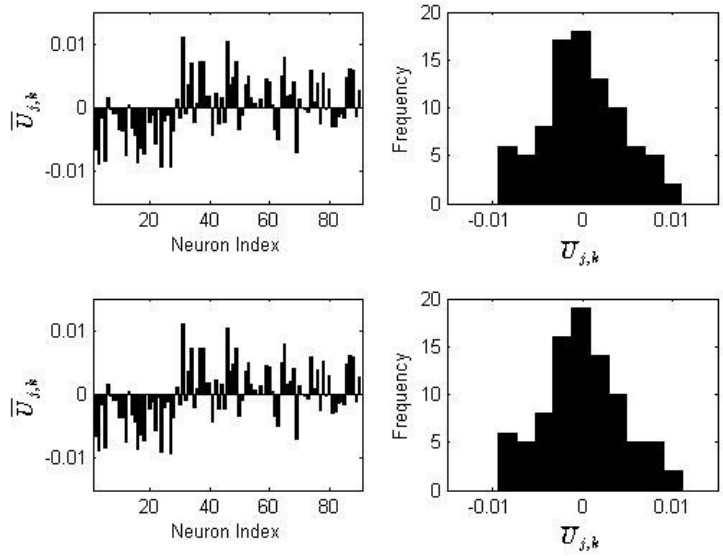

Figure 4.50: Histogram of VTS results using the watch interrupt model with network and neural plasticity with a sparse network. Here the data is equally distributed around zero.

4.6.3 Event-driven Simulation: VTS Large Interrupt Model

The maximally-connected network as shown in Figure 4.51 and sparsely-connected network depicted in Figure 4.55 have quite different heart rate responses and these simulations may be roughly divided into three regions: (i) region 1 where only network

plasticity is active over $0 \leq t \leq 500$ (maximally- and sparse-connected networks), (ii) region 2 where network plasticity and neural plasticity are active over the range $t = 500$ to $t \sim 4500$ (sparsely-connected network) and $t = 500$ to $t \sim 7500$ (maximally-connected network) , and (iii) region 3, approximately $t > 4500$ (sparsely-connected network) and $t > 7500$ (maximally-connected network), wherein only network plasticity is active. Region 2, the zone of autonomic derangement (details in Section 4.2.2) is considerably larger here (about 4000 s for the sparsely-connected and about 7500 s for the maximally-connected) than that seen in the previous subsections for the sequential simulation (about 300 s) or the VTS watch interrupt model (about 700 s). The significant prolongation of the autonomic derangement (Region 2) occurs in the large interrupt model due to the tendency to update specific neurons over others. This tendency can be quantified, for example, by simply noting that the average peak updating in the large interrupt model provided in the first column of Figure 4.54 is approximately double that seen in Figure 4.46 (note that the Java sequential simulation has a limiting constant mean updating).

It is interesting that the length of region 2 when autonomic derangement is active is considerably longer for the maximally-connected network than the sparsely-connected network. A possible explanation is that there is an increased number of predominant cells in the maximally-connected network that take up a larger fraction of the updating than that which occurs in the minimally-connected network.

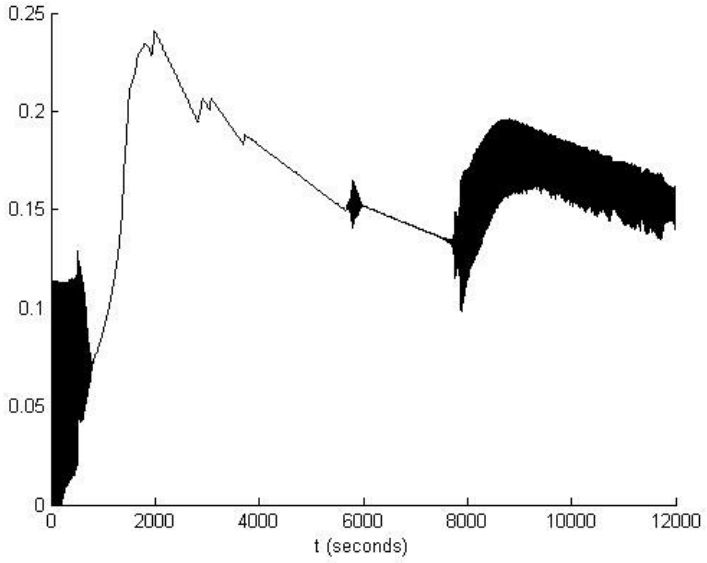

Figure 4.51: Effect on heart rate using the VTS large interrupt simulation with network and neural plasticity. This network is a maximally-connected network. Scaled heart rate (dimensionless) is on the y-axis. Network plasticity is active for t < 500 seconds. Both network and neural plasticity are active from t = 500 seconds to t = 7500 seconds. Then, only network plasticity. Compared to the other simulations, the large interrupt model show large changes in dynamics within region 2.

The fractional contribution to synaptic weights is quite similar for maximally and sparsely-connected networks in Figures 4.52 and 4.56. Again, these results differ from those seen for the sequential simulation with a progression in the large change interrupt model toward a decreasing arrangement in fractional contribution magnitude with the middle predominant as seen in the previous section where neural plasticity is turned off.

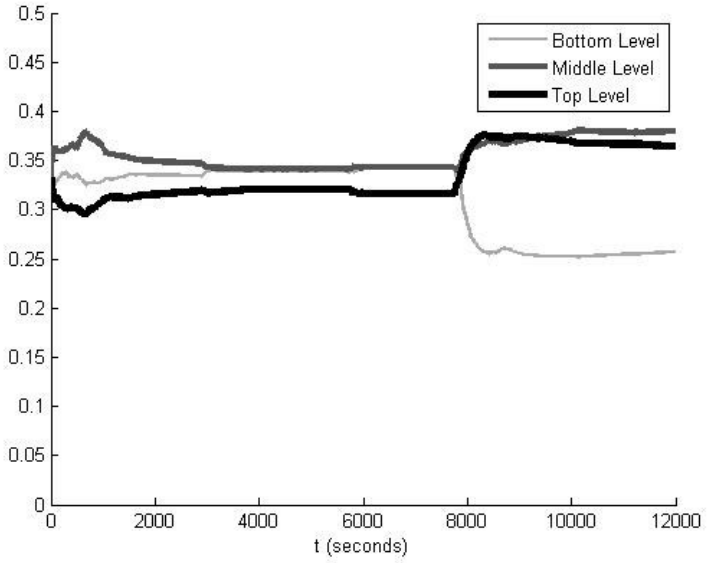

Figure 4.52: Average neighbor weights using the VTS large interrupt model with a maximally-connected network. Average neighbor weight is on the y-axis. There is little dynamic response until neural plasticity is turned off at $t \sim 7500$ seconds.

Again, we note that neural updating in the maximally-connected network considered in Section 4.5 where neural plasticity is turned off and presented in Figure 4.15 is quite similar to that seen here in Figure 4.54. Once again, this appears to indicate that neural plasticity does not have a significant impact on the statistics of neural updating.

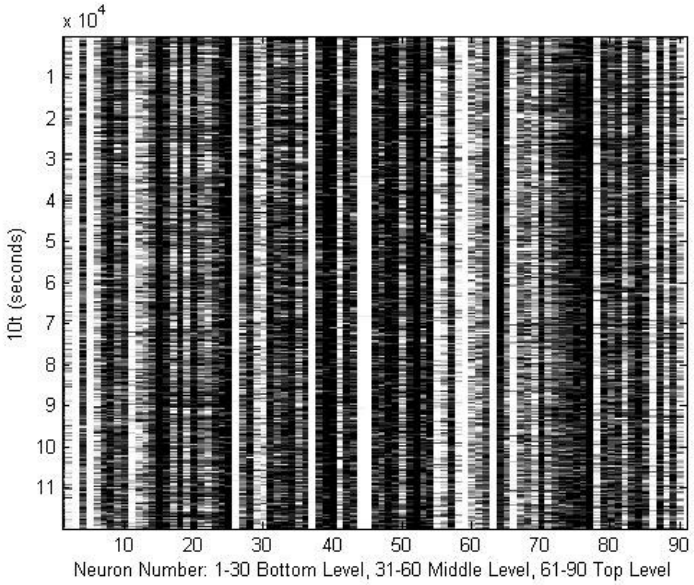

Figure 4.53: Neural network for VTS large interrupt model using network and neural plasticity. This network is a maximally-connected network. There appears to be uniformness in neuron layer activity for this simulation.

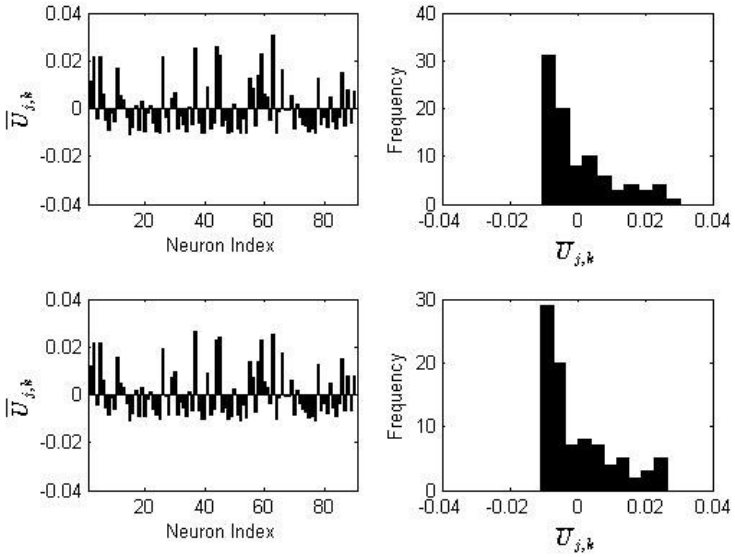

Figure 4.54: Histogram of VTS results using the large interrupt model with network and neural plasticity with a maximally-connected network. Variation here is right-sided.

Finally, the robustness seen in the previous simulations for the watch interrupt model is observed again. In particular, the large change interrupt model heart rate response for the maximally-connected network shown in Figure 4.43 and sparsely-connected network in Figure 4.47 show similar heart rate responses unlike the sequential simulation maximally-connected network in Figure 4.39 and sparsely-connected network Figure 4.40 that are quite different.

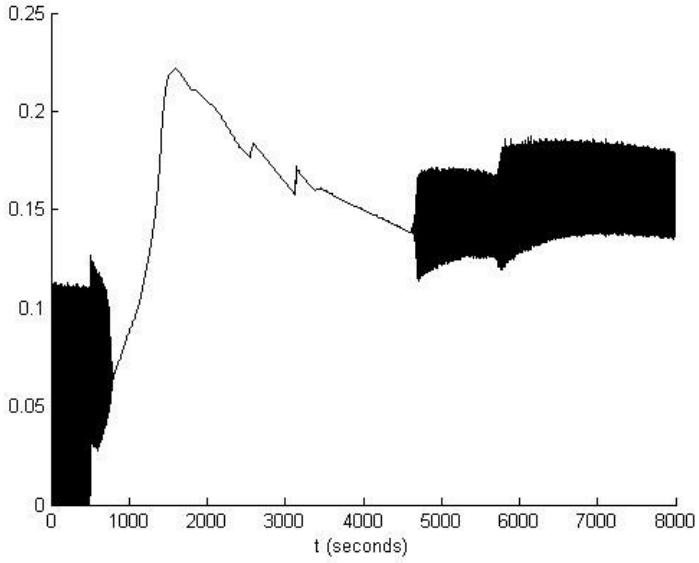

Figure 4.55: Effect on heart rate using the VTS large interrupt simulation with network and neural plasticity. This network is a sparse network. Scaled heart rate (dimensionless) is on the y-axis. Network plasticity is active for t < 500 seconds. Then, both network and neural plasticity are active until about $t = 4500$ seconds. Only network plasticity is active from $t > 4500$ seconds. Results are comparable with the VTS watch interrupt model.

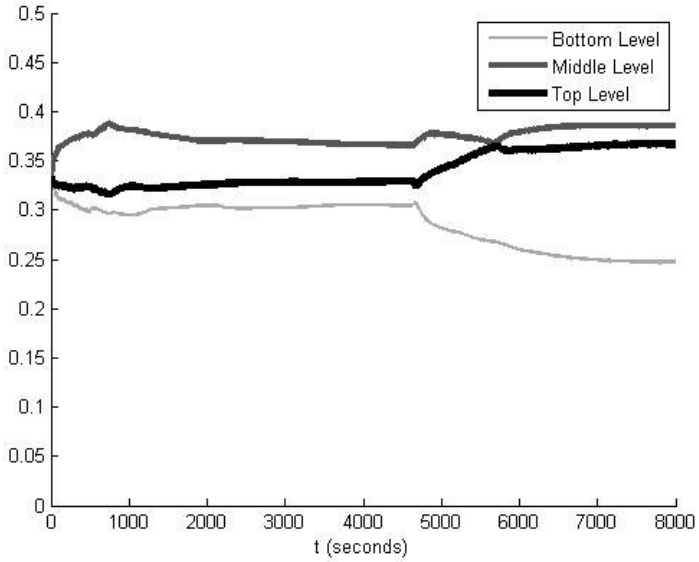

Figure 4.56: Average neighbor weights using the VTS large interrupt model with a sparse network. Average neighbor weight is on the y-axis. Results are complementary to the VTS watch interrupt model.

4.6.4 Summary

The response of the neural network in the presence of autonomic derangement showed distinctive similarities and differences for the sequential and event-driven simulations.

The heart rate responses were quite similar during the autonomic derangement for all simulations (save for the Java sequential simulation for the sparsely-connected network) differing mainly in the magnitude and peak during the derangement phase of the simulations. The main difference between the sequential and event-driven simulations lies with the sensitivity to the network connectedness: the event-driven simulations show little effect compared with the sequential simulation that show a relatively large effect. The implication of the observed sensitivity to network connectedness of the sequential simulation in the presence of derangement suggests that the event-driven simulation confers a level of robustness to parametric change. We would not be able to discover this robustness unless we considered simulating the model in an event-driven way. This is an obvious statement but leads to the conclusion that timing effects in nonlinear models may be an important component that should be considered.

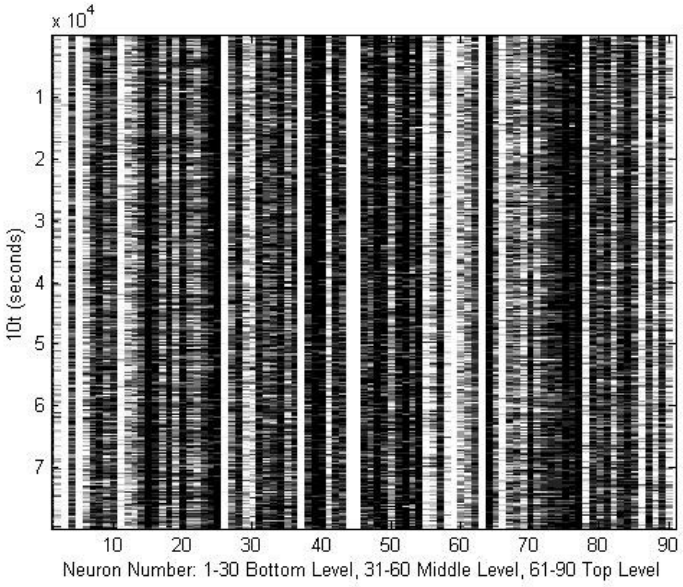

Figure 4.57: Neural network for VTS large interrupt model using network and neural plasticity. This network is a sparse network. As with the VTS large interrupt model with maximally-connected network, there appears to be uniform activity amongst layers in the simulation.

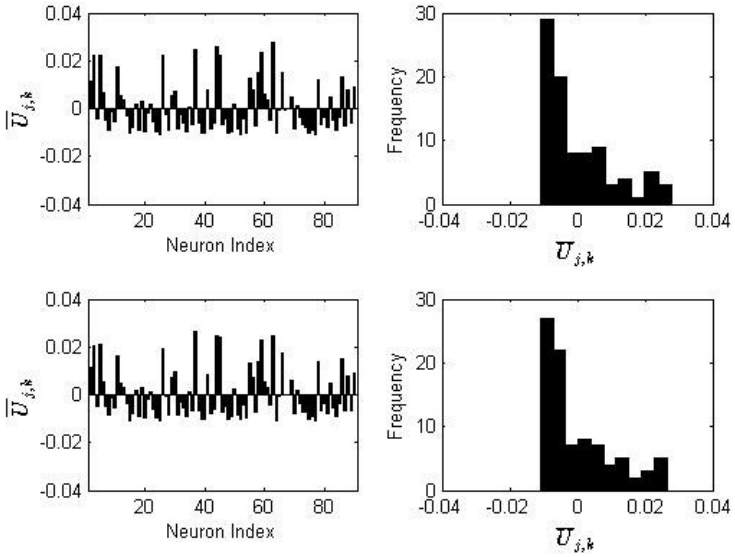

Figure 4.58: Histogram of VTS results using the large interrupt model with network and neural plasticity with a sparse network. Variation here is right-sided like in the maximally-connected simulations using the large interrupt model.

CHAPTER 5 CONCLUSION

The goal of this thesis was to examine how sequential and event-driven simulations differ for neural-networked cardiac control that begins from the same initial conditions in the presence of an ischemic event (heart attack). The ischemic event was initiated at the same time in each simulation and the heart rate response was examined with respect to the presence or absence of network plasticity (changes in synaptic weights) and neural plasticity (autonomic derangement). From the clinical perspective these networks may be regarded as *twins* that are identical. In other words, we considered a single network structure for each of the maximally-connected and sparsely-connected networks. The heart rate response to the ischemic pathology was therefore considered as a response that a *twin* would make to a pathological event.

The results in Kember *et al* [1] are the sequential (Java) simulations in this work and there is no change in the conclusions made in Kember *et al* [1]. Specifically, the central conclusion in that work was that heart rate is a highly complex function of what occurs within the network and this makes the relationship between heart rate and what goes on inside the network murky at best. This *murkiness* implies a level of complexity in network processing that may account for some of the unexplained variability in heart rate that currently limits its usefulness as a clinical marker [1]. That conclusion still stands. However, the results of this thesis also show that if one attempts to understand closed-loop control that occurs within a network, such as the one considered here, then orders of operation should be considered. In particular, the results in Kember *et al* [1] were based on the assumption that orders of operation could be ignored. Hence the conclusions in Kember *et al* [1] could not be extended to include the larger point observed here that changes in the orders of operation can significantly improve or worsen the progression of a pathology.

The differences that we observed between the sequential and event-driven simulations stem from sensitivity to orders of operation that includes changes in the dynamics and the selection of steady-states. Note that attempting to *match* a model such as this to what is observed in clinical practice is not possible because it would require measuring the

activity of cells within the network. This dependence of steady-states upon orders of operations is due to the nonlinearity of the mathematical model coming from, for example, constraints placed on neuron activity, state-dependence of changes in synaptic weights (network plasticity), and so on. Finally, we would like to state that there is no *best* simulation method. Rather, it is more useful to think of the event-driven simulation technique as introducing extra parameters into the mathematical model and that this introduction that leads to an infinite set of solutions in the sense that new steady states are observed. Although it is not considered here, it is likely that the link between sequential and event-driven simulations is singular because the former do not possess infinite solutions.

We observed significant changes in heart rate responses during and after the ischemic event, for example, heart rate level and the presence/absence of oscillations in heart rate. Since heart rate level and the presence/absence of oscillatory behaviour is clearly of clinical importance, the work considered here suggests that using mathematical models to form clinical conclusions with respect to actual treatment is difficult at best. This point may be extended further to animal studies that are intended to improve our understanding of cardiac pathology and its evolution. Mathematical models are considered to be *fitted* when the predictions from the mathematical model match the measured data under a criterion that measures the difference between the model and the measured data. These approaches are entirely predominant in the literature yet they may be neglecting an important dependence of model dynamics coming from order of operations.

In this thesis we also considered two methods of event-driven updating of neural properties. The first is a representation of a *domino-cascade* that we called the *watch interrupt model* wherein a neuron is updated if at least one of its neighbouring neuron's activity levels changed. The second method gave precedence to the neurons experiencing the large changes in activity over those neurons experiencing relatively smaller changes in activity and this was termed the *large change interrupt model*. These methods do not appear to exist in the literature but considerable research is underway in genetic

simulations to examine the effect of various interrupt models and these may be a useful addition.

A literature review was conducted on event-driven simulations for biological networks. Although event-driven simulation research is currently underway in many areas, the biological/neurological applications are mainly focused on simulation of granule cells [6], spiking neural networks [5] and genomics. Other than the research completed by Kember *et al* [1] (and previous Kember *et al* work [22-24]), the area of comparison between sequential and event-driven neurocardiac control is entirely novel. It is also important to note that the event-driven simulations presented in this thesis do not make use of approximation tables [6,8,10] to formulate results.

Nonlinearities are frequently seen in mathematical models of biological systems. Since both the steady-state and dynamics are altered through changes in orders of operation in the simulation of nonlinear models, the standard approach to *best-fit* such models ignoring orders of operation may be a meaningless exercise. Finally, the variability that emerges from orders of operations in mathematical models is clearly dependent upon the model details and therefore may provide a greater understanding as to how biology utilizes and manages what appears to be *noise*.

BIBLIOGRAPHY

[1] G. Kember, J. A. Armour and M. Zamir. Neural control hierarchy of the heart has not evolved to deal with myocardial ischemia. *Physiol. Genomics*, 45:638-644, 2013.

[2] C. De Michele. Optimizing event-driven simulations. *Computer Physics Communications*, 182:1846-1850, 2011.

[3] P. Valentini and T. E. Schwartzentruber. A combined Event-Driven/Time-Driven molecular dynamics algorithm for the simulation of shock waves in rarefied gases. *Journal of Computational Physics*, 228:8766-8778, 2009.

[4] E. Lerner, G. Düring and M. Wyart. A loss-event driven scalable fluid simulation method for high-speed networks. *Computer Networks*, 54:112-132, 2010.

[5] L. Caron, M. D'Haene, F. Mailho, B. Schrauwen and J. Rouat. Event management for large scale event-driven digital hardware spiking neural networks. *Neural Networks,* 45:83-93, 2013.

[6] R. R. Carrillo, E. Ros, S. Tolu, T. Nieus and E. D'Angelo. Event-driven simulation of cerebellar granule cells. *Biosystems,* 1:10-17, 2008.

[7] A. A. Prinz, V. Thirumalai and E. Marder. The functional consequences of changes in the strength and duration of synaptic inputs to oscillatory neurons. *J. Neurosci*, 23(3):943–954, 2003.

[8] K. Roeller, S. Herminghaus and A. Hager-Fingerle. Sinusoidal shaking in event-driven simulations. *Computer Physics Communications*, 183:251-260, 2012.

[9] M. Girardi. Event-driven simulations of insulating grains in an external electric field. *Physica A: Statistical Mechanics and its Applications*, 389:4520-4527, 2010.

[10] R. R. Carrillo, E. Ros, B. Barbour, C. Boucheny and O. Coenen. Event-driven simulation of neural population synchronization facilitated by electrical coupling. *Biosystems*, 87:275-280, 2007.

[11] Y.A. Omelchenko and H. Karimabadi. Event-driven, hybrid particle-in-cell simulation: A new paradigm for multi-scale plasma modeling. *Journal of Computational Physics*, 216:153-178, 2006.

[12] M. Höst, B. Regnell, J. Natt och Dag, J. Nedstam and C. Nyberg. Exploring bottlenecks in market-driven requirements management processes with discrete event simulation. *Journal of Systems and Software,* 59:323-332, 2001.

[13] B. Denkena, S. Kršninga and K. Doreth. Operational Planning of Maintenance Measures by Means of Event-driven Simulation. *Procedia CIRP*, 3:61-66, 2012.

[14] P. Grube, F. Núñez and A. Cipriano. An event-driven simulator for multi-line metro systems and its application to Santiago de Chile metropolitan rail network. *Simulation Modelling Practice and Theory*, 19:393-405, 2011.

[15] K. Matsukawa. Central command: control of cardiac sympathetic and vagal efferent nerve activity and the arterial baroreflex during spontaneous motor behavior in animals. *Physiol Exp.*, 97:20-28, 2011.

[16] M. LaRovere. Heart rate and arrhythmic risk: old markers never die. *Europace*, 12:155-157, 2010.

[17] J. T. Cacioppo, G. G. Berntson, P. F. Binkley, K. S. Quigley, B. N. Uchino, and A. Fieldstone. Autonomic cardiac control noninvasive indices and basal response as revealed by autonomic blockades. *Psychophysiology*, 31:586-598, 1994.

[18] P. B. Corr, K. A. Yamalla, and F. X. Witkowski. Mechanisms controlling cardiac autonomic function and their relation to arrhythmogenesis. *System, The Heart And Cardiovascular*, Raven, New York, 1986.

[19] A. Malliani, P. J. Schwartz, and A. Zanchetti. Neural mechanisms in life-threatening arrhythmias. *J. Am Heart*, 100(5):705-15, 1980.

[20] M. Horackova, R. P. Croll, D. A. Hopkins, A. M. Losier, and J. A. Armour. Morphological and immunohistochemical properties of primary longterm cultures of adult guinea-pig ventricular cardiomyocytes with peripheral cardiac neurons. *Cell, Tissue*, 28:411-425, 1996.

[21] M. Horackova, J. A. Armour, and Z. Byczko. Distribution of intrinsic cardiac neurons in whole-mount guinea pig atria identified by multiple neurochemical coding. A confocal microscope study. *Cell Tissue Res.*, 297:409-421, 1999.

[22] J. A. Armour. Cardiac neuronal hierarchy in health and disease. *Physiol., Am J.*, 287:262-271, 2004.

[23] G. Kember, J. A. Armour, and M. Zamir. Neural control of heart rate: The role of neuronal networking. *J. Theoret. Biol.*, 277:41-47, 2011.

[24] G. Kember, J. A. Armour, and M. Zamir. Dynamic neural networking as a basis for plasticity in the control of heart rate. *J. Theoret. Biol.*, 317C:39-46, 2012.