A flexible suite of programs for modelling the cortex with a mean-field scheme

A thesis submitted in partial fulfilment
of the requirements for the degree of

Master of Engineering

in Physics and Electronic Engineering
at the
University of Waikato
by

Yuan-Kuei (Jay) CHANG

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2007

Acknowledgements

I would like to use this opportunity to acknowledge all the people that had help me through my masters. Special thanks to my supervisor Dr. Marcus Wilson. Thank you very much for attending to my queries thoroughly with patience. I am sure some of the questions I got for you must sound silly, but you have always made it easy for me to ask them. Your door was always open for me, and the ease for me to trouble you without making prior arrangements really meant a lot to me. Writing a thesis was not an easy task, I am grateful for your guidance throughout the way.

Thank you Dr. Alistair Steyn-Ross for your help for programming. It was easier to construct this software system with your support. Thank you Dr. Michael Cree for assistance with LaTeX. The layout of this thesis gets much better with the thesis template and the help you provided.

Last but not least, I would like to express my gratitude to my family and all my friends for all your support, without your support and encouragement, completing my masters will be a great hardship for me. Your positive and joyful comments kept me until the end.

Abstract

The cerebral cortex contains many neurons. The neuron is part of the nervous system and it receives and transmits the electrical signals. These signals are significant to a human's behaviour. Since the neurons are charged, these charges produce electrical fields, so these neural signals can be measured by using scalp electrodes in electroencephalography (EEG). As long as the brain is not dead, the spontaneous activities of neurons will produce a series of EEG signals.

There are many models that have been developed for simulating the cortical signal, and mostly each model is focused towards a different purpose or application. Often, a different computer code has to be written for each different application, and this can be inefficient. Therefore, this project aims to develop a software system for simulating cortical signals where the model used for the system can be changed easily. Furthermore, the system is requested to be versatile and easy-to-use for many applications.

The developed system is written in MATLAB in response to a user requirement and mostly applies to any model which uses a mean-field approach. Only the specific inputs need to be modified for changing the model. This thesis details how this system is developed. The main limitation of the system is computational resources, much the same as other cortical modelling. However, all the user requirements had been satisfied. The system can simulate the response of the neurons for any condition and generate simulated EEG data to the user. The user can analyze the cortical activities using the standard signal processing techniques such as a power spectrum. This software is very helpful for the research of sleep and anaesthesia.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Biological Background

A human's behaviour is controlled by the brain, and the cerebral cortex controls many of these, such as thinking, communicating, reasoning and consciousness (1). The cerebral cortex is a thin layer containing about 100 billion neurons and 100 trillion synapses (2). Neurons are nerve cells, which are responsible for transmitting the electrical signal. The structure of a typical neuron is basically the same as any other cell, except there are few specialized structures make a neuron different to other kinds of cell. One key difference between a neuron and any other normal cell is that a neuron has many extensions. These extensions usually look likes branches or spikes extending out from the cell body. One of these extensions is called axon, and the others are called dendrites. Dendrites are responsible for receiving chemical messages from other neurons and axon is to transmit an electro-chemical signal to other neurons. Sometimes the axon is over a considerable distance (It can be 90cm long for a human). There is a very tiny gap between the axon ending and the dendrite of the next neuron. It is called the synapse (or synaptic gap, or synaptic cleft). There are 1000 $\sim$ 10,000 synapses for each neuron. Figure 1.1 shows the diagram of neuron structure. The surface of the axon contains hundreds of thousands of miniscule mechanisms called **ion channels** and these ion channels cause the signals by an electrochemical process. These neural signals control human's behaviour. Since these signals are electrical signals, so they can be easily detected with scalp electrodes of an electroencephalogram (EEG). The EEG detects the electrical field caused by the neuron's charge distribution. However, the electrical field that a single neuron creates is too weak and it is too difficult to be detected by a scalp electrode, so the scalp electrodes can only detect the EEG signal if the following two factors are both satisfied (5):

- There must be large amount of neurons firing.
- The electrical field that every specific neuron creates must be large enough

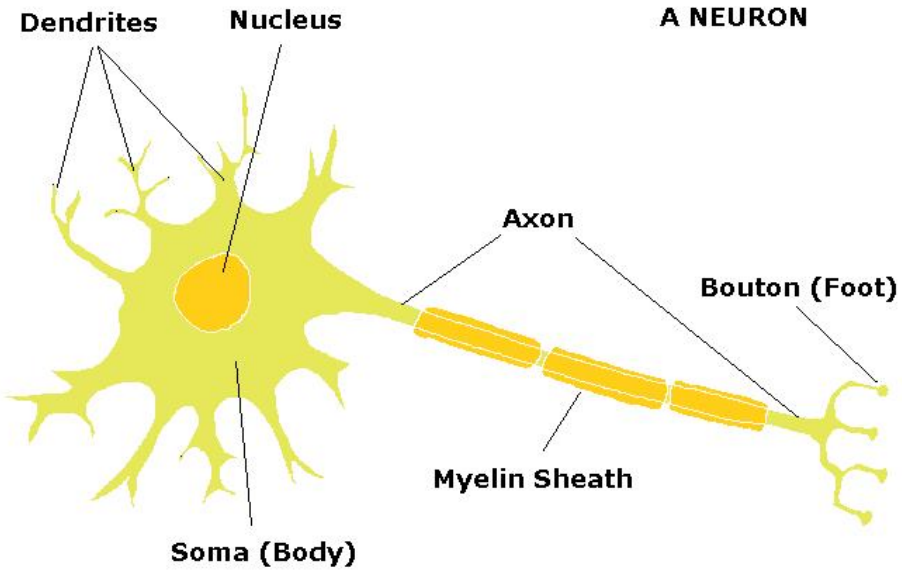


Figure 1.1: *Diagram of neuron structure. Source: Ref. (4)*

when close to scalp.

The EEG signal changes for different behaviours or circumstances.

1.2 Modeling Approach

Currently, there are four methods for modelling the neurons in the human's cerebral cortex:

Neural Networks This builds a network for neurons and each individual neuron in the network will be simulated.

Monte-Carlo A simulation which uses statistical sampling techniques to obtain the solution by a probabilistic approximation. The complete data sets will be generated for each simulated neuron (6).

Mean-field It contains a set of ordinary differential equations, where these equations are developed by approximation based on averaging techniques.

Population-Density statistical behaviour is concerned for a group of neurons rather than individual neurons.

No matter which method is used, the main limitation in mathematically modeling populations of interacting neurons is computational resources (time and memory).

For example, the case **8×8 hypercolumn-model for V1 with 64,000 neurons** takes 1 day to simulate 4 seconds real time (7) (by Jim Wielaard and collaborators, Columbia)

Neural Networks simulate every neuron in the network, but there are 100 billion neurons in the cerebral cortex and the work for modeling will be massive, so it is impossible to simulate the whole cortex. However, it provides a detailed simulation and it is useful to simulate a small region in the cerebral cortex. The accuracy of Monte-Carlo depends on the number of random samples taken, so it is very accurate if there are large amount of samples have been taken. However, it might lose the accuracy if the amount of samples taken is not sufficient. The calculations for modelling based on Monte-Carlo are very complicated either. It will take a long time before the software based on this method is robustly developed. It takes too much computational resources and it is not efficient. Therefore, the advantage of both neural networks and Monte-Carlo is high accuracy, but their disadvantage is low efficiency. This makes mean-field and population-density methods more convenient. The computational techniques within both methods have provided time saving alternatives compared to neural networks and Monte-Carlo, but approximations must be made which compromise the accuracy available. However, both mean-field and population-density methods get less efficient as the underlying neuron model is made more realistic and complicated due to the increase in number of state variables (8). No matter what, they are still much faster than neural networks and Monte-Carlo nowadays. The mean-field method is the first choice for this project, because the implementations in mathematics and computational techniques are easier for mean-field than population-density and the cortical modelling group in the University of Waikato has a history of using mean-field models. Therefore, the system for this project will be designed to be used with the common mean-field models.

1.3 Uses of the mean-field approach

The continuum models with mean-field approach for modelling the neurons are introduced by Freeman (9) and Nunez (10). The mean-field models are concerned about a population (a continuum) of neurons rather than single neurons. The model based on this form for modelling a group of neurons sampled by a single scalp electrode have been developed by Wright and Liley (11), Robinson et al. (12), Liley et al. (13), and Rennie et al. (14). The Liley model has been modified by Steyn-Ross et al. (15) with implementation in two spatial dimensions by Wilson et al. This modified model is useful for studying EEG phenomena, such as:

- Sleep (e.g. Slow Wave Sleep ⇔ Rapid Eye Movement Sleep transition)
- Anaesthesia (e.g. Conscious ⇔ Unconscious transition)

The code included in this thesis is developed based on this modified Liley model. However, one of the purposes for designing this software system is to allow the user to change the model easily, so the user can implement many different models using a mean-field approach.

1.4 Macrocolumn

The system will be designed to model averages of properties, across macrocolumns. The human cortex consists of a number of macrocolumns. Each macrocolumn is a volume of ∼1 mm area by ∼1 mm thick and each one consists of around 100000 neurons. There are broadly two kinds of neuron: excitatory and inhibitory neurons. The excitatory neurons encourage neurons to fire and the inhibitory neurons discourage neurons. A schematic diagram of a macrocolumn is shown in Figure 1.2.

1.5 Aim of the Project

The main objective for this project is to design the software with a clean format of code, and it must be easy for the user to change the mean-field model used by the system. Therefore, the aim of the project is to design a software system, which is:

1. versatile / widely applicable
2. easy to use
3. multi-applications

The detailed requirements will be specified and discussed in Chapter 2.

The system could be used for academic or clinical research. It can be used to examine, for example, the response of a simulated human cortex to an anaesthetic or natural sleep.

1.6 Thesis Structure

Since this thesis is for Master of Engineering, not for Science, the thesis will focus on the design logic of software development for this project, rather than the underlying science principles. However, in place, it will be necessary to explain some of the science involved.

Chapter 2 lists the specifications for the software. They are derived from the potential users, such as cortical modelling researchers, and each user requirement has been detailed explained and discussed. Possible solutions have been discussed for

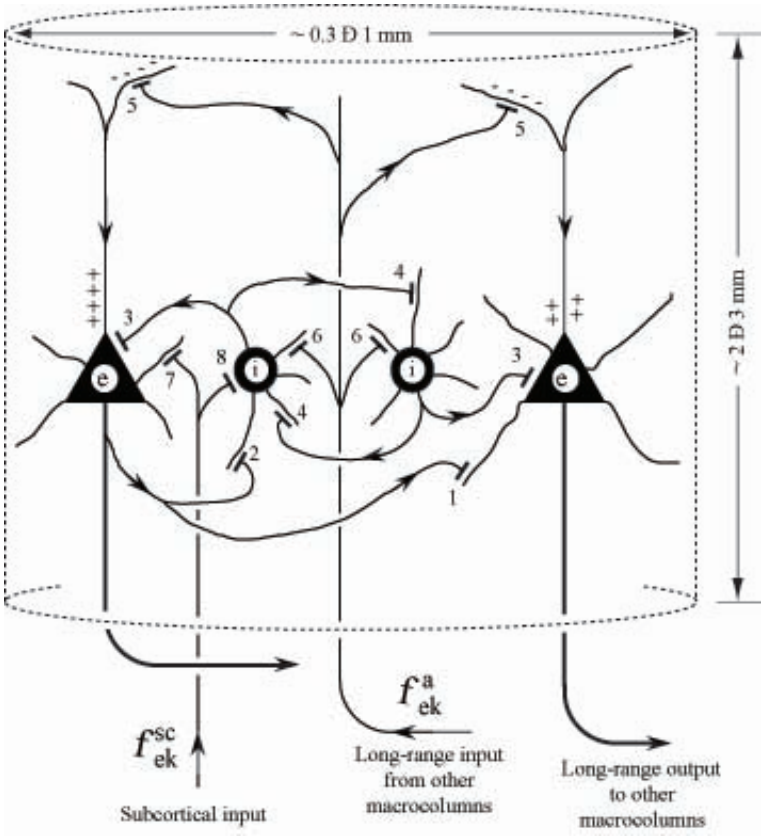


Figure 1.2: *Schematic of a Macrocolumn. Source: Ref. (16)*
The diagram above explains the connections within a macrocolumn. Connection 1 is the connection between excitatory neurons. Connection 2 is the connection from excitatory neuron to inhibitory neuron. Connection 3 is from inhibitory to excitatory. Connection 4 is from inhibitory to excitatory. 5 and 6 are the connections from other macrocolumns. Connections 7 and 8 are the subcortical inputs.

each requirement also. The software will be designed based on these specifications. Every user requirement has been ranked with priority (in Appendix B) by D.A. Steyn-Ross and M.T. Wilson. This project will try to meet the requirement with the higher priority first while designing the software. The goal is to meet all the user requirements for this software system.

Chapter 3 is the top-level system design. The prototype of this software system is designed and discussed in this chapter. This prototype is designed based on the user requirements and possible solution of each requirement was discussed in Chapter 2. The structure of this software system is presented and discussed in this chapter.

Chapter 4 is the detailed system design for the function required by the simulation. All the functions and their relationship to others will be discussed in this chapter. The purpose for each function will also be explained in detail here. This chapter shows how each function was designed, and science principles / theories used for each function.

Chapter 5 is the detailed system design for the functions required by the data processing. This explains the mathematics and techniques involved in processing the data of the cortical signals, and also the programming implementation for developing the functions.

Chapter 6 shows the results of this project. The final product is displayed and discussed in this chapter. The final product of the software is tested by several standard cases and each testing case is showed here. The plots and data outputed by the system are explained in this chapter also, so the purpose and use of these outputs can be identified here. The computer code is in the Appendix C and provided on a CD-Rom with this thesis.

Chapter 7 evaluates this software. The user requirement is discussed again here. Unlike the discussion in Chapter 2, the discussion in this chapter focuses on the accomplishment of the user requirement as a whole. The accomplishment of this project is examined here and also the reasons for its success and failures.

Chapter 8 discusses the improvements which can be made for the software. The future work for this software are listed and discussed here. It concludes with future research topics to provide improvements to increase the efficiency and practicability of the software.

Chapter 9 is a summary of the conclusions made from the design of this software. This chapter discribes how this project has gone, and the overall achievement of this project. The possible applications of this software have been discussed in this chapter also.

Chapter 2

Requirement

There are two fundamental types of software product: Generic products and Customised (or bespoke) products (17). The generic products are stand-alone systems that are produced by a development organization and the end user can buy them from the market. The customized products are developed by a software contractor particularly for a certain group of users due to their request. Since this software system will be used for academic research purpose and only the particular users will use this software, so this software system should be classified to be customized product. For a customized product, it is quite important to define the user group and discuss with them about their requirement for a new software system before start designing. The users are expected to be the cortical modelling researchers, mostly the staff and students of the University of Waikato. Therefore, the requirements for this software are produced by discussing with them about their demand for the system. The full documents user requirement is in Appendix B. Each user requirement will be discussed briefly in this chapter. The possible solution and the way of design are also included for each requirement. These requirements will determine the specification of the system, and also provide the idea about the way how the system will be designed.

The following requirements are specified by M.T. Wilson and D.A. Steyn-Ross. Most of them directly relate to the structure or the performance of the system. These requirements are significant and they provide ideas about how the system will be designed. The design of the system based on these requirments will be discussed in the next chapter.

1. *Must run on PC and MAC*
   Since most users are familiar with Windows and there are also many programmers and developers use MAC, so they would like this system to be compatible for both operating systems.
   Requirement Evaluation:
   This requirement makes the system to be more convenient to be used by the

users. This requirement is not be as necessary as others, but it will be satisfied if possible.

Possible solution:

This system will be written in a programming language, which can be ran on both Windows and Mac.

2. *Must be easy for a user to modify.*

   No one would like to use a program that is difficult to understand how to use it in the first place. It will take a lot time in learning for the first-time user and it is not efficient, so a nice and friendly user-interface is usually essential for designing a software tool. Since the cortical model for the system contains many equations, parameters, etc... so the user requests that these must be organized in a logical way and represented with an easy-to-use format.

   Requirement Evaluation:

   This requirement relates to the complexity of using the software and it is significant to the user, so it will be satisfied.

   Possible solution:

   The inputs will be kept in separate input functions or files, so the user can modify them easily.

3. *Full documentation, including a user-guide, must be written.*

   The cortical model for the system involves a lot equations and parameters, so the final product of the design might be complex and it might not be easy for the first-time user to use. Therefore, the user requires an user-guide of this software system with step-by-step tutorial in it.

   Requirement Evaluation:

   An user-guide will let the user to understand the system quicker and better, so this requirement is significant and will be satisfied.

   Possible solution:

   Due to the demand, this design will have an user-guide with step-by-step tutorial in it. The user-guide will be included in Appendix A.

4. *User must be able to supply any model that can be written as a set of first-order differential equations in the form* $dy = f(y)dt + \xi$*, where* y *is a multidimensional vector,* f *is a function,* t *is time, and* ξ *is a noise-input.*

   This is a fundamental requirement based on the form of the common mean-field models (e.g. Liley model) in which the behaviour of the cortex is expressed in terms of a series of first order differential equations in time. The equations of some models might be nonlinear or complicated function of time. There is generally no analytic solution for this kind of equations, so a numerical integration method will be used for the design. Therefore, the user requests a system that a set of first-order differential equations can be supplied by the

user to the system.

Requirement Evaluation:

Since the system is intended to be used for a mean-field modeling approach, so this is the essential requirement and it directly relates to the structure of the software. It will be satisfied; otherwise the system might not be the one as user expected.

Possible solution:

These equations will be stored in a separate input file. Therefore, the user will probabily need to do some Mathematics work on paper before changing the equations for the model, if those equations are not in a form of first-order differential equation.

5. *Up to 50 first order equations possible*

   There might be many equations for a model, so the user would like that there is no limit or a high limit on the number of the input first-order differential equations. The desirable number of equations that the system can accept should be up to 50.

   Requirement Evaluation:

   This requirement relates to the limit of the model. The number of equations that the system can accept should be designed to as many as possible; otherwise it might not be able to use some models that involve lot equations. However, it does not affect the function of the system, so this requirement is not compulsory to the design, but the system will be designed to satisfy this requirement.

   Possible solution:

   The number of the equations is not a problem for the design, as long as they are in a form of first-order differential equation when input to the system. The system will be designed to store and numerical integrate these equations, so there should be no limit for the number of the equations. However, more input equations to the system means it involves more calculations and takes more computational resources, so it slows the simulation speed. Therefore, the user will be warned by the user-guide about this.

6. *Model should work in at least two spatial dimensions i.e. so we can implement equations where each component of y is itself a matrix in space.*

   The user would like to see the dynamic signal on the cortex, where the cortex should be assumed to be 2-D or 3-D in space. Therefore, the system is requested to be able to simulate the cortex in at least two spatial dimensions.

   Requirement Evaluation:

   This requirement relates to the realism of the simulation. The user can understand the variation of the cortical signal in space better if it is a multidimensional simulation, so this requirement is essential and will be satisfied.

Possible solution:
Since this system is going to be designed for simulating the cortex and the cortex is assumed as a 2-dimensional square plane (e.g. as for the modelling of Wilson et al. Bojak and Liley [17]), so the variables of the equations for the model will be integrated as two dimensions in space.

7. *User should be able to change the equations (model)*
   The user might want to change the model for the system or modify / update the equations in the future, so the user would like to be able to access the equations in the system and the equations can be modified by the user.
   Requirement Evaluation:
   This requirement is compulsory for the system, because there are many different neural models. The user should be able to change the model for the system. The system will be designed to be able to use any mean-field model and this requirement will be satisfied.
   Possible solution:
   The equations for the model will be stored in a separate input file and the user can access this file easily. Therefore, if the user would like to change the model for the system, then it can be done by replacing or modifying the equations in the equations input files.

8. *User should be able to change values of any parameters*
   The user wants to see the response of the system by using different parameters value, so it is requested that the user must be able to access the parameters stored in the system and their values can be changed by the user.
   Requirement Evaluation:
   This requirement is essential, because the user needs to see the response of the cortical signal for different circumstance, so the system will be designed to allow the user to do it.
   Possible solution:
   The parameters will be stored in a user input file and the user can also easily access this file. If the user would like to change value of any parameters, then it can be done by replacing the value for the parameter in this file.

9. *User should be able to specify how different parameters change with time*
   Some parameters might vary with time, so the user would like to be able to access the system to specify the way of the parameters changes with time.
   Requirement Evaluation:
   This requirement relates to the uses of the system, so this requirement is significant and will be satisfied.
   Possible solution:
   This can be done by adding a function changing the parameters with time,

where it needs the user to write into the function about how the parameters changing with time. There will be a control-flag for selecting a changing parameter, so the user can select this for the simulation easily. Furthermore, the user can modify this function by adding a new way of parameter changing or modifying the original ones.

10. *User should be able to specify a perturbation in the system at some point(s) in space and time*
    The user would like to see how the system responses if there is a step-change in any of the variables, so the user requests that the system should be able to do this. The user should to be able to control and specify it in the system.
    Requirement Evaluation:
    This requirement is not as essential as the previous one, but this requirement also relates to the uses of the system. Therefore, this requirement will be satisfied.
    Possible solution:
    This can be done by adding a function to the system to perturb the system in a user-defined manner with a user-specified position and time. There will be a control-flag for this, so the user can easily activate or deactivate the perturbation. And the user can change the manner of the perturbation by modifying this function.

11. *Software should be able to output, at the user's request:*
    Requirement Evaluation:
    The following requirements relate to the complexity for analyzing the simulation results. The user can understand the cortical signal better if more information is provided. Therefore, the system will be designed to satisfy the following as many as possible.

    - *A data-file containting $y_j(r,t)$ where j is any user-specified component of y, r is space and t is time*
      The user might want to know how a variable changes with time, so a system is requested to output and save the data of variables after the simulation.
      Possible solution:
      The variables data of the simulation will be saved by the system. The user can recall the saved file to access the data for the chang of variable in space and time.
    - *Movie file of $y_j(r,t)$ in a standard format*
      The user might want to see a dynamic picture of a variable. A system is requested to output a movie file for the selected variable in a standard file format, so the movie file can be opened by a general video player program

of a PC.

Possible solution:

The system will record the frames of the plot for the variable and generate the movie file. However, some users might not be able to open the movie file if it is saved in a rare file format that needs to be opened by a specific program or player. Therefore, it will be saved to a common file format and can be opened by the most video player programs (something like in Mpeg or Avi format).

- *Formatting plots of $y_j(r_a, t)$ where j is any user-specified component of y and r_a is some user-specified point in space*

  The user might want to see the time variation of a variable during the simulation for a point on the cortex plane, so the system is requested to generate a temporal plot for a variable on a point of the cortex plane and the user should be able to specify this point.

  Possible solution:

  A function will be designed to generate the plot for a user-specified variable in a user-specified position. This will provide the information about the fluctuation of a variable. The user can do the further analysis based on this.

- *A power spectrum (power against temporal frequency) for the system*

  The user might want to know the strength and frequency components of the output, so the system is expected to output a plot for the power spectrum.

  Possible solution:

  There will be a function designed to define the power and temporal frequency of the simulated signal, and a 2-D plot will be generated by the function based on these.

- *A spectrogram plot (power against frequency and time)*

  The user might want to see how the frequency changes with time for a signal, so the system is expected to output a spectrogram plot.

  Possible solution:

  The possible way to do this is to divide the data into several sections with a certain time length, and then defines the power and frequency for each section.

- *Plots of power against temporal frequency and spatial frequency*

  The user might want to know how the signal strength changes with frequency and space, so the system is expected to output a plot for this.

  Possible solution:

  A 3-D plot will be generated by the function. The amplitude, frequency and position of power fluctuation can be seen on the plot.

- *Stationary (equilibrium) state(s) of the system, and their stability, as a function of time*

  The stationary states are the solutions for the equations of the model ($\frac{dy}{dt} = 0$) when there is no noise. The user might want to know how the stationary states change with time, and also the stability of the stationary states, so the system is expected to output the data for these.

  Possible solution:

  There will be a function designed to define the stationary states of the system for every time-step in the simulation, and also the stability of the stationary states. However, this will delay the simulation speed, so a control flag for this will be added to the function. The user can either activate or deactivate this function according to his requirement.

12. *No third-party software required for which the University of Waikato does not already possess (and intends to keep renewing) a license.*

    Since the most users for using this system will be mainly the staff and students in the University of Waikato, so the system must be able to be ran using programs for which the University of Waikato expects to have a license.

    Requirement Evaluation:

    This requirement relates to the cost of using this system. It does not affect the performance of the system, so it is not as significant as other requirement. But it will be satisfied if possible.

    Possible solution:

    The programming tool for designing this system will be the one that the University of Waikato already has a license for.

13. *User should not need to change the main code in order to run the software*

    The user would like a system that can be used easily. The user does not want a system that it requires the user to modify the main code for running the software, because it increases the complexity and difficulty for using the system. Therefore, the main code of the system is expected to be fixed.

    Requirement Evaluation:

    This requirement does not affect the performance of the system, but it relates to the complexity of using the system. Therefore, this requirement will be satisfied if possible, to make the system as an easy-using software.

    Possible solution:

    The main function of the system will be designed to be fixed. The user will only need to access the input functions for using the system.

14. *Should be easy to modify the code to use different numerical integration methods*

    The user might want to change the numerical integration method for the sim-

ulation, so the user would like to be able to access for this and it is expected to be easy for the user to modify.

Requirement Evaluation:

Some numerical integration methods are accurate, but take time. Some methods are faster, but not very accurate. The user sometimes needs a faster simulation, or sometimes a more accurate simulation. So this requirement might not be as significant as some other requirements, but it relates to the uses of the system and will be satisfied if possible.

Possible solution:

There will be more than one numerical integration methods and they will be put into the system. There will be a control flag designed to select the method for numerical integration. The user can also write a new integration method if required.

15. *No restrictions on the use of the software by the University of Waikato for academic research purposes.*

    The system is expected to be used freely by the University of Waikato for academic research.

    Requirement Evaluation:

    Requirement Evaluation: This requirement is essential, because the main users are the staff and students in the University of Waikato.

    Possible solution:

    The copyright and intellectual property of this system belongs to the University of Waikato. There will be no restriction for the staff and students in the University of Waikato to use this system for academic research purpose.

16. *Ensure, where feasible, that all numerical routines can be tested against standard* ***calibration*** *suite.*

    There might be mistakes in the input numerical routines or bugs in the system. The user would like the system to be designed that they can be tested standard cases, so the user can define and locate the mistake or bug easily and quickly, and correct them.

    Requirement Evaluation:

    This requirement does not affect the performance of the system, but it lets the user to be able to debug the system. It is useful and this requirement will be satisfied if possible.

    Possible solution:

    There will be a testing mode for the function with numerical routines, and the testing mode will contain one or more testing cases with different standard values for the input parameters, so the user can compare the output results by the function with the ideal theoretical results. There will be a control flag designed to activate or deactivate the testing mode. The user will also be able

to switch the testing cases. This will be useful for finding any bugs.

Chapter 3

Top-level System Design

3.1 Programming Language

There will be a lot of equations and mathematical calculations involved in the finished product, so a programming tool for developing this software must be powerful in dealing with mathematics. Some programming languages such as Fortran or C++ are quite common nowadays and they can do basic mathematical calculations, but the calculations for the simulation here involves matrix calculations and complex numbers. These make the programming languages such as Fortran or C++ less suited to be used for designing this software. Other programming languages such as Mathcad, Mathematica and MATLAB are quite powerful in dealing with mathematics and plotting. They can be installed in both Windows and MAC. However, the MATLAB is already well used at the University of Waikato, and the users (mainly the staff and students at the University of Waikato) are familiar with MATLAB. Therefore, MATLAB will be the first choice of the programming languages used for developing this software.

3.2 Software Process Models

This system is totally software based. It is mostly developed from scratch, but some functions were developed previously by D. A. Steyn-Ross and M. T. Wilson. There are many process models for the software development, such as (17):

The waterfall model This processes the fundamental process activities of specification, development, validation, and evolution as separate phases (e.g. requirements specification, software design, implementation, testing and so on).

Evolutionary development This approach is based on interleaving the fundamental process activities of specification, development and validation. An ordinary system is rapidly developed from basic specifications. A completed system which satisfies the user's request will then be developed after refining

its ordinary system with user requirement.

Component-based software engineering This approach innovate the existent reusable components. The components required by the system will be mainly developed by integrating the existent ones rather than developing them from scratch.

Each process model is used for processing different kind of software development. Since this is an individual project and based on the user requirement in the previous chapter, the predicted final structure of this software system will not be large, so the waterfall model can be used here for the process model of the software development. Therefore, the stages for developing this software will be:

1. Requirement analysis and definition
2. System and software design
3. Implementation and unit design
4. Integration and system testing
5. Operation and maintenance

3.3 System

The requirement is discussed in the previous chapter. Based on the requirements and possible solutions in the previous chapter, this software will contain two sections in its code:

1. Simulation (This contains three stages: input, main and output stage.)
2. Output Processing and Analyzing

The block diagram shown in Figure 3.1 is the prototype of this software:

Since the equations of the model inputted to the system will be in the form $\frac{dx}{dt} = f(x,t)$, so the main steps in the simulation are:

1. Find equilibrium states x_0 such that $\frac{dx_0}{dt} = 0$
2. Find their stability to small perturbations
3. Integrate forward in time — so if $x(t)$ is known, then $x(t + \Delta t)$ can be found

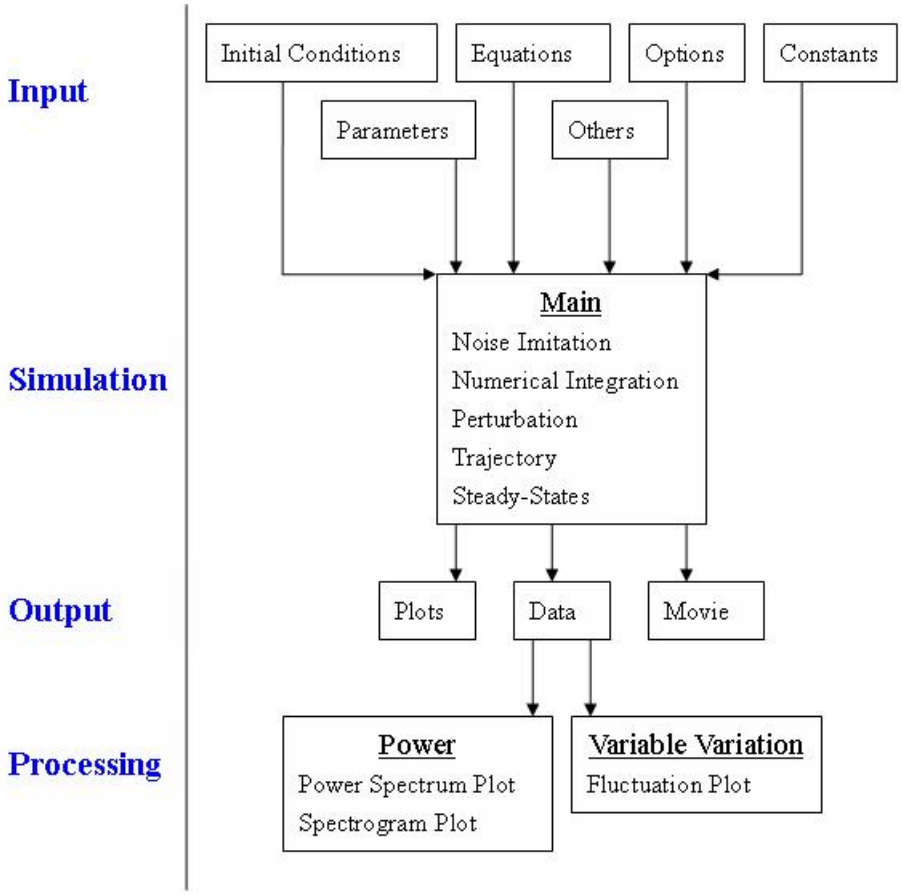


Figure 3.1: *Block diagram of the system*
The system takes the inputs (parameters, control-flags, equations, etc...) into the main function for simulating the cortical signals, then outputs the data, plots and movie of the simulation. The simulation data can be further processed by the output processing functions (`function Power_Spectrum` & `function Variable_Variation`).

3.3.1 Input Function

There are many things must be input to the system for the simulation, and most of them are for the user to define, so they will be stored in the input functions and files. Since these inputs have different uses in the system, so the system of this software will be designed to contain several input functions and the user can easily modify the part of the input for the simulation. These input functions will store constants, user's options, parameters, initial conditions, etc... These inputs can be classified into two groups: Generic Input and Specific Input.

Generic Input

These are associated with the generic operation of the simulation and they are independent to exact equations of the model. The changes in these generic input

functions will change the operation of the simulation, such as total simulation time steps or time step-size of the simulation. These inputs are:

Constants These are the numerical constants required by the equations of the model while being numerical integrated. The values for these numerical constants are fixed all the time and the user should not need to change any of these values for the simulation. E.g. length of cortex, membrane rate-constants etc...

User's option (Control Flag) There are many functions contain many different conditions for the simulation. These control-flags are for the user to select the condition from those with a simple input as 1 or 0 (true or false), so the user will not need to modify the codes for selection of the output. E.g. selection for numerical integration method, whether a movie is saved, etc...

Parameters (independent to the model) These are the figures independent to the model and require the user to change or check their values every time before starting simulation. E.g. Δt (time step-size), `tend` (total simulation time steps)

Trajectory There are some parameters, which might change with time during simulation, e.g λ_i (a scaling of the IPSP), γ_i (the inhibitory rate constant). This function is for the user to input how these parameters change with time (maybe equations).

Perturbation The user might want to see the response of the system if there is a step-change on some variables or parameters. This function is for the user to input this variable / parameter and the position, time and amount for the occurrence of this step-change

Specific Input

These inputs are associated with the model of the simulation (i.e. the exact equations that the user chooses to model the cortex). They must be modified if the model is changed. These inputs are:

Equations (First-Order Derivatives) These are for numerical integration of the variables. In the function the user specifies the first order derivatives of each variable in the system. These must be changed when the particular mean-field model is changed.

Stationary states (Initial Conditions) The method for finding the solution to $\frac{dx}{dt} = 0$, where y is any variable. These can be the initial values ($x(t = 0)$) for the variable before starting numerical integration.

Stability The equations for defining whether the stationary states are stable or not, and how stable / unstable.

Parameters (model dependent) These are the numerical constants dependent to the model and require the user to change or check their values every time before starting the simulation. E.g. ΔV^{rest} (effect of altering extrasynaptic ion channels), λ_{Ach} (effect of muscarinic neuromodulation).

Noise It is the subcortical input and it is simulated by random numbers.

Example of Cortical Equations

The model used in the code to illustrate the system is based on Liley model, but modified by Steyn-Ross et al. (16). The modified Liley model contains several equations, and they are:

$$\tau_e \frac{dV_e}{dt} = V_e^{rest} - V_e + \lambda_{Ach}\rho_e\psi_{ee}\Phi_{ee} + \lambda_i\rho_i\psi_{ie}\Phi_{ie} \tag{3.1}$$

$$\tau_i \frac{dV_i}{dt} = V_i^{rest} - V_i + \lambda_{Ach}\rho_e\psi_{ei}\Phi_{ei} + \lambda_i\rho_i\psi_{ii}\Phi_{ii} \tag{3.2}$$

$$(\frac{d^2}{dt^2} + 2\gamma_{ee}\frac{d}{dt} + \gamma_{ee}^2)\Phi_{ee} = \gamma_{ee}^2(N_{ee}^{\alpha}\phi_{ee} + N_{ee}^{\beta}Q_e + \phi_{ee}^{sc}) \tag{3.3}$$

$$(\frac{d^2}{dt^2} + 2\gamma_{ei}\frac{d}{dt} + \gamma_{ei}^2)\Phi_{ei} = \gamma_{ei}^2(N_{ei}^{\alpha}\phi_{ei} + N_{ei}^{\beta}Q_e + \phi_{ei}^{sc}) \tag{3.4}$$

$$(\frac{d^2}{dt^2} + 2\gamma_{ie}\frac{d}{dt} + \gamma_{ie}^2)\Phi_{ie} = \gamma_{ie}^2(N_{ie}^{\beta}Q_i + \phi_{ie}^{sc}) \tag{3.5}$$

$$(\frac{d^2}{dt^2} + 2\gamma_{ii}\frac{d}{dt} + \gamma_{ii}^2)\Phi_{ii} = \gamma_{ii}^2(N_{ii}^{\beta}Q_i + \phi_{ii}^{sc}) \tag{3.6}$$

$$(\frac{\partial^2}{\partial t^2} + 2\upsilon\Lambda_{ee}\frac{\partial}{\partial t} + \upsilon^2\Lambda_{ee}^2 - \upsilon^2\nabla^2)\phi_{ee} = \upsilon^2\Lambda_{ee}^2 Q_e \tag{3.7}$$

$$(\frac{\partial^2}{\partial t^2} + 2\upsilon\Lambda_{ei}\frac{\partial}{\partial t} + \upsilon^2\Lambda_{ei}^2 - \upsilon^2\nabla^2)\phi_{ei} = \upsilon^2\Lambda_{ei}^2 Q_e \tag{3.8}$$

where firing-rates $Q_{e,i}$ is the function of soma potential $V_{e,i}$. The equations of Q_e and Q_i are:

$$Q_e(V_e) = \frac{Q_e^{max}}{1 + exp[-\pi(V_e - \theta_e)/\sqrt{3}\sigma_e]} \tag{3.9}$$

$$Q_i(V_i) = \frac{Q_i^{max}}{1 + exp[-\pi(V_i - \theta_i)/\sqrt{3}\sigma_i]} \tag{3.10}$$

And ψ_{ee}, ψ_{ei}, ψ_{ie} and ψ_{ii} are the weighting functions dependent upon the soma potentials.

$$\psi_{ab} = \frac{V_a^{rev} - V_b}{V_a^{rev} - V_b^{rest}} \tag{3.11}$$

Table 3.1: The standard parameters required by the modified Liley model

Variable / Parameter	**Description**	**Standard Value**
$V_{e,i}$	Soma Potential	—
$Q_{e,i}$	Firing Rates	—
$\Phi_{ee,ei,ie,ii}$	Postsynaptic Flux	—
$\phi_{ee,ei}$	Presynaptic Flux	—
$\tau_{e,i}$	membrane time constants	0.04, 0.04 s^{-1}
$Q_{e,i}$	maximum firing rates	30, 60 s^{-1}
$\theta_{e,i}$	sigmoid thresholds	-58.5, -58.5 mV
$\sigma_{e,i}$	standard deviation for threshold	4.0, 6.0 mV
$\rho_{e,i}$	gain per synapse at resting voltage	0.001, -0.00105 $mV \cdot s$
$V_{e,i}^{rev}$	reversal potentials at synapse	0, -70 mV
$V_{e,i}^{rest}$	cell resting potential	-64, -64 mV
N_{ea}^{α}	long-range e to e or i connectivity	3710
N_{ea}^{β}	short-range e to e or i connectivity	410
N_{ia}^{β}	short-range i to e or i connectivity	800
$\langle\phi_{ea}^{sc}\rangle$	mean e to e or i subcortical flux	750 s^{-1}
$\langle\phi_{ia}^{sc}\rangle$	mean i to e or i subcortical flux	1500 s^{-1}
γ_{ea}	excitatory synaptic rate constant	300 s^{-1}
γ_{ia}	inhibitory synaptic rate constant	65 s^{-1}
$L_{x,y}$	spatial length of cortex in model	500 mm
a_{mc}	area of macrocolumn	1 mm^2
Λ_{ea}	characteristic inverse length-scale for connections	0.2 mm^{-1}
υ	mean axonal conduction speed	1400 mms^{-1}
ΔV_e^{rest}	Effect of altering extrasynaptic ion channels	-2.0 to +2.0 mV
λ_{Ach}	Effect of muscarinic neuromodulation	1.0 to 1.5
λ_i	Multiplier on inhibitory synaptic weight	1.0

where a and b here stand for either excitatory or inhibitory. The Table 3.1 shows the variables and parameters required by the modified Liley's equations.

The modified Liley's equations define how the variables change with time and the relationship between variables and other terms. Equation 3.1 and 3.2 are the first-order differential equations Equation 3.3 $\sim$ 3.8 are second-order differential equations in time. Equation 3.1 and 3.2 describe how the soma potential changes with time. The term "$\rho\psi\Phi$" shows that the soma potential will be affected by the synaptic inputs, and $V_{e,i}^{rest} - V_{e,i}$ is a decay term for the soma potential. Equation 3.3 $\sim$ 3.6 describe the time variation in synaptic inputs for the postsynaptic fluxes, where the term $(\frac{d^2}{dt^2} + 2\gamma_{ee,ei,ie,ii}\frac{d}{dt} + \gamma_{ee,ei,ie,ii}^2)$ shows the time variation. Equation 3.3 and 3.4 have one extra term more than 3.5 and 3.6: $N_{ee,ei}^{\alpha}\phi_{ee,ei}$. This is because the excitatory neurons have long range connections with other neurons and the inhibitory neurons don't have. The equations 3.7 and 3.8 describe the long range (e.g. one part of the cortex to another) propagation of presynaptic fluxes, where the term

$(\frac{\partial^2}{\partial t^2} + 2\upsilon\Lambda_{ee,ei}\frac{\partial}{\partial t} + \upsilon^2\Lambda_{ee,ei}^2 - \upsilon^2\Delta^2)$ describes the standard wave.

Since these differential equations are non-linear, so they are not likely to have an analytical solution, which is why they need to be simulated rather than solved. This requires an update rule that maps from $y(t)$ to $y(t+\Delta t)$, where Δt is time step-size. It will be discussed in detail later.

The parameters and user's options (control-flags) will be stored in the same function, because they are the inputs that the user needs to access the most for using this software. This can make this software more convenient for a user to use. The constants will be saved in an independent file, because they are mostly fixed and it is not usual for a user to change them. The other inputs, such as equations, stationary-states, trajectory, etc... will be saved into their independent functions, so the user can quickly locate the part they want to modify and modify the inputs or change the model easily.

Since the user requests the model to work in at least two spatial dimensions, so the variables in the equations of the model will be assigned with an $N \times N$ matrix at the start of the simulation. This $N \times N$ matrix corresponds to the cortex plane (the simulated flattened cortex).

3.3.2 Main Function

The main function of the system should be fixed and the user does not need to modify the main code while using the software. Since this system does not take many actions for the simulation, so the structure of the main function can be simple and pithy. The main job for the main function is to numerically integrate the equations to simulate the signals. Therefore, it will basically recall all the input functions, fixed functions and output functions for the simulation and loop them as required. This system will be designed to contain two numerical integration methods, second-order and fourth-order Runge-Kutta algorithms modified for use with a stochastic signal (see e.g. Kloeden and Platen 1992 (18)). Both methods will be coded in the same function and there will be a control-flag in the function for switching the methods, so the user can switch the methods easily by changing the control-flag in the user-input function. The main function will read-in this control-flag and determine the method to be used for numerical integration. The data of the simulation will be outputed by the output functions.

3.3.3 Output Function

The output function of the system will record the data of the variables, steady-states, stability, movie and plots of the simulation, and save them to a file, so the

user can open the file and analyze the data without running the simulation again.

3.4 Output Processing Function

This system will be designed to contain two functions for processing the output data. One is for power spectrum and another one is for variable variation.

3.4.1 `Function Power_Spectrum`

This function will load the data-file saved by the main function and calculate the power strength, temporal and spatial frequencies of the signal, then generate the plots based on these. The plots generated by this function will be:

2-D Power Spectrum Power against temporal frequency

3-D Power Spectrum Power against temporal and spatial frequencies

Spectrogram Plot Power against frequency and time

3.4.2 `Function Variable_Variation`

This function will load the data-file saved by the main function (time variation) and plot the fluctuation of the signal, so the user can see how the signal changes in time and space during the simulation.

Both functions will load the output data of the main function and generate the plots. The user can analyze the results based on these plots, or do further analysis of his own, if required.

Chapter 4

Detailed System Design — Simulation

One of the objectives for this project is to design a system with a clear code. The user can easily use this software without understanding the code of the whole system, and can also change the model easily without producing a different version of code.

The code can be divided into two sections:

1. Simulation
2. Data Processing

The simulation part will be discussed first in this chapter. Figure 4.1 is the structure of the simulation.

The simulation contains three stages: the first is the input stage, the second is the main stage and the third is the output stage. There are several input functions in the system; this is to allow a logical grouping of input variables, flags and functions. It defines the method of the simulation, and also the model of the system. It does take some time for the user to check the content of the input functions before starting the simulation, but this is necessary and cannot be ignored. The input functions will be called by the main function, then the main function outputs the data to the output functions for display and saving.

4.1 `function Cortex_System`

This is the main function of the system. The content of this function is basically the procedure of the simulation. This function calls all the functions required by the simulation, and also loops the functions to numerical integrate step by step for the variables. This function is designed to be fixed and the user does not need to

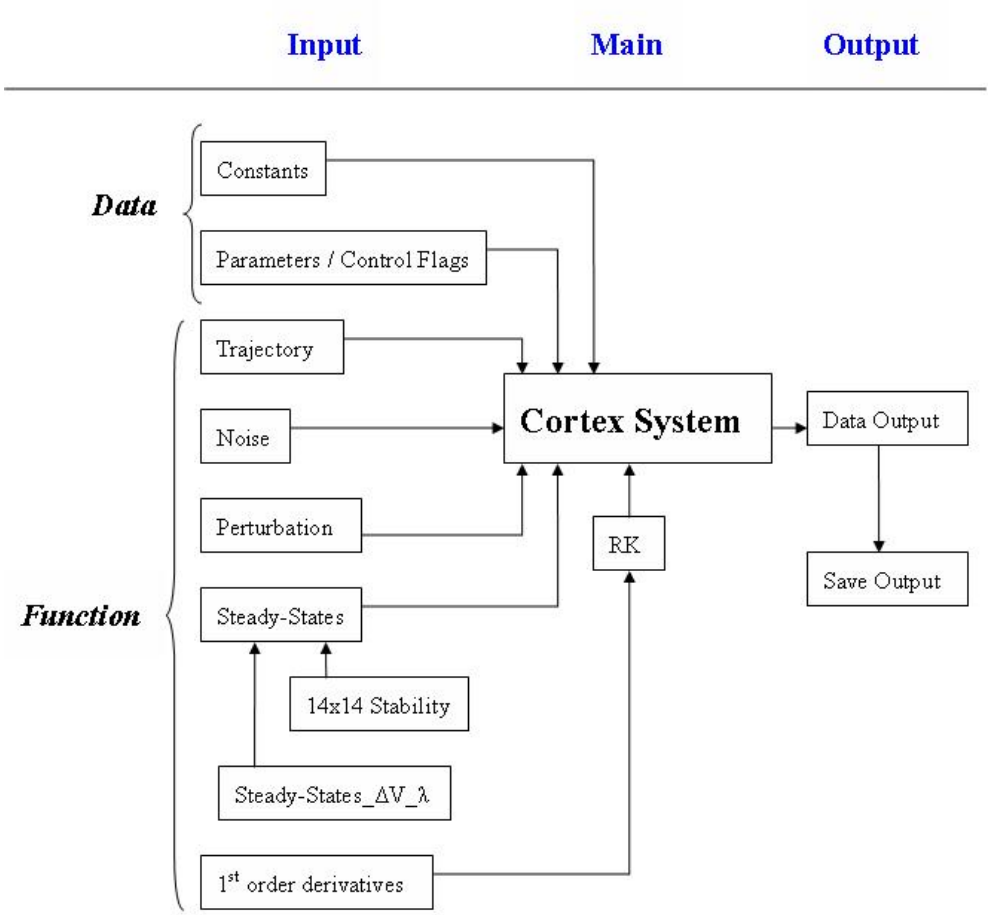


Figure 4.1: *Structure of the Simulation*
The main function `Cortex_System` *calls the input data and functions and performs Runge-Kutta numerical integration to simulate the signals, then outputs the simulation data for saving.*

change anything in this function while using this software. Since that this is a fixed function, a clear and simple code is required. Anything that needs the user to define or anything unnecessary will not appear in this function. Figure 4.2 is the flowchart of this function.

Step 1

There are two functions containing input parameters. They are `function init_hasselmo_globs` and `function Input`. `Function init_hasselmo_globs` contains constant parameters related to the model, e.g. reversal potentials, resting potentials, etc... It is from the historical form of Alistair Steyn-Ross' code, which looked up constants that had been defined in a paper by Hasselmo et al. (19). The user does not need to modify this function for the simulation. `Function Input` stores user-defined parameters. The user needs to check or modify the parameters value every time before starting the simulation. The main function reads these parameters in and set them to global (for other functions to recall).

Step 2

Once the input parameters have been read in, the system calls `function find_steady_states` to find the equilibrium solution for the equations. This is usually used as the initial condition. The user will have to amend the `function find_steady_states` when the equations of the model are changed. The initial conditions are defined according to the user's input parameters value. The initial conditions will be returned to the main function and the simulation is now ready to start.

Step 3

The loop starts here after the initial conditions have been defined. The system calls the trajectory function (`function find_trajec`) to define the trajectory that the user selected, where the **trajectory** here means how the parameters change with time. For example, if an anaesthetic is modeled, we would expect the magnitude of the IPSP (Inhibitory Post-Synaptic Potential), modeled by ρ_i in Liley's equations, to change with time. Since all trajectories are time-dependent, so the parameters must be updated by this function if there is a trajectory. Therefore, this function is called by the system at the start of the loop to feed the system with the updated parameters.

Step 4

This system models the subcortical input with noise, so the system calls the noise function here to imitate noise. Since the signal in the cerebral cortex always has noise in the real world, and the noise of the cortex is probably meaningful and significant to brain (20). Therefore, a good noise imitation is important and the system must take the noise into the consideration while simulating the signal, because white

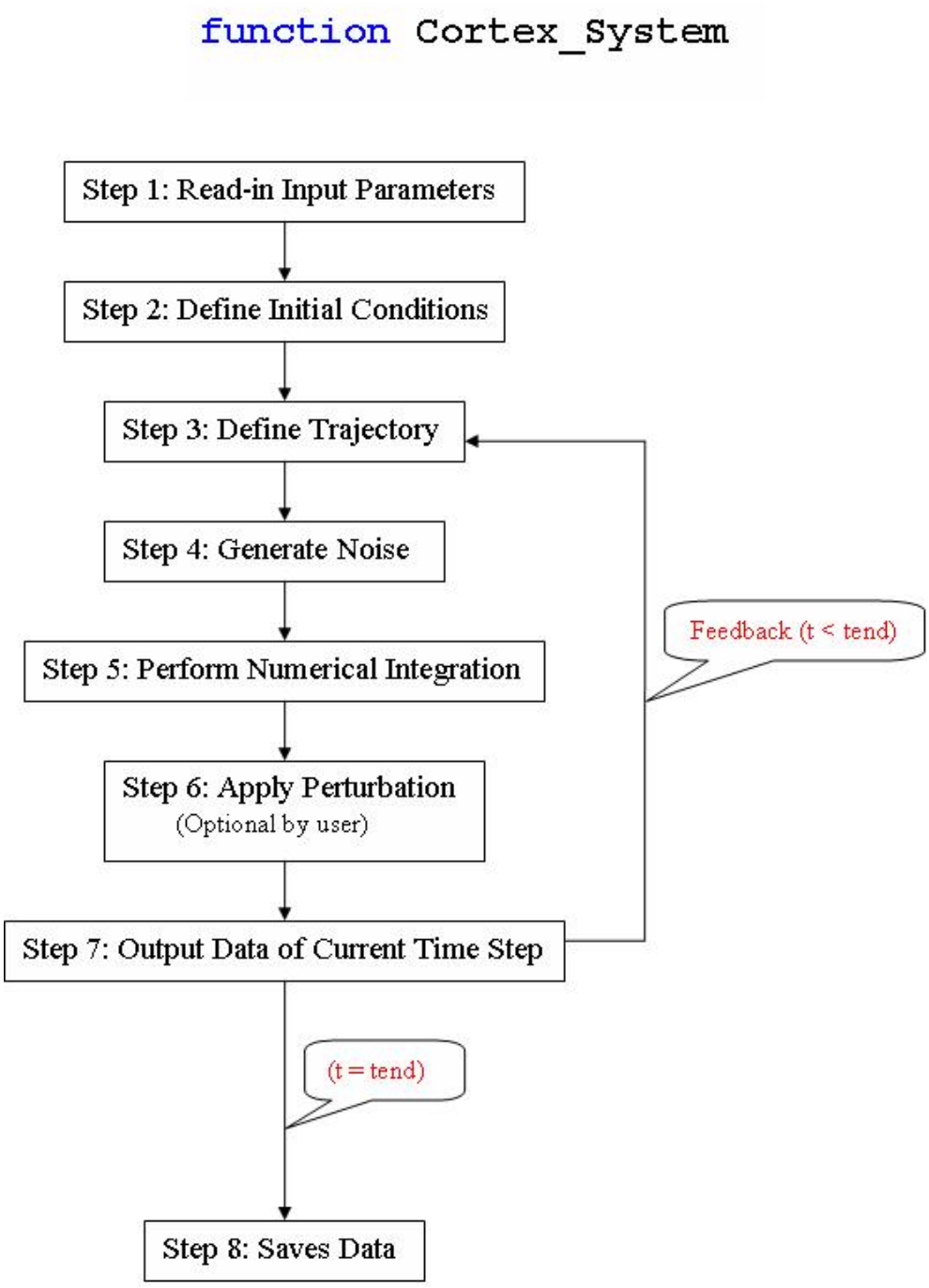


Figure 4.2: *Flowchart of the main function*
where `t` *is the current time step and* `tend` *is the total time steps.*
The main function `Cortex_System` *loops Step 3 ∼ Step 7 to numerically integrate the variables for the simulation. The loop is ended when* `t = tend`*, and then the simulation data will be saved.*

noise is usually used to make the simulation more convenient. The noise function is called here, because the noise imitation must be done before numerically integrating the variables. The noise function is independent of the trajectory function, so the order of these two functions is not significant, but these two functions must be called before numerical integration.

Step 5

After the trajectory is defined and the noise is generated, the variables will be integrated at this step. The initial condition of the variables will be recalled here at the first time-step; otherwise the variables of previous time-step will be recalled to numerically integrate for the current time-step. The integrated variables of current time-step will be returned back to the system for the next round of the simulation. The numerical integration method that this system uses for integrating the variables is based on Runge-Kutta algorithm, and the function used for that is `function Runge_Kutta_Integration`. This function is designed to be fixed and the user does not need to modify anything in this function for simulation or changing model.

Step 6

The perturbation function is called here by the system. It is the user's option for activating or deactivating the perturbation. If the perturbation is activated, then this function will perturb the system in a way defined by the user. If the perturbation is deactivated, then this function will not do anything and it will take no effect for the simulation.

Step 7

This is the last step for a loop. The simulated data of the current time step will be outputed to the global for collection if the user activates the saving before running the simulation. The graphs of the current time step for the user-selected variable, usually V_e, will also be plotted and displayed here. If the current time step is not yet the final time step, then the procedure will go to Step 3 and start another round of simulation. If it is the last time step of the whole simulation, then it will go to Step 8 to end the simulation.

Step 8

If the data-saving or movie-saving is activated by the relevant control flags, then it will be saved to a file at this step. Since the data has been sent to global at step 7 for collection, so the collected data with the whole simulation time steps will be recalled from global to the function (`function save_output`) for saving. If both data- and movie- saving are not activated, then there will be nothing to be saved, and this function will take no effect in the simulation. The simulation ends at this step.

4.2 `function Runge_Kutta_Integration`

Some differential equations can be solved exactly. However, this code is designed to solve nonlinear differential equations. There is, in general, no analytic solution for nonlinear differential equations. The variables value can only be predicted rather than solved in this case. Therefore, a numerical integration must be used for prediction. Some of the common numerical integration methods are (18): Euler, second and fourth-order Runge-Kutta. The Euler method is the fastest while predicting the values for the variables in the simulation, but it is often not stable and it can not provide an accurate result for Liley model. Therefore, the numerical integration method we used to integrate the variables is based on Runge-Kutta algorithm only. This function contains two numerical integration methods: second-order and fourth-order Runge-Kutta algorithms. The equations for RK2 and RK4 are:

Second-order RK updating rule:

$$y_{n+1} = y_n + \frac{\Delta t}{2} \cdot [K_1 + K_2]$$

$$y_n = \frac{dy}{dt}$$

$$K_1 = F(y_n)$$

$$K_2 = F(y_n + \Delta t \cdot F(y_n))$$

Fourth-order RK updating rule:

$$y_{n+1} = y_n + \frac{\Delta t}{6}[K_1 + 2K_2 + 2K_3 + K_4]$$

$$K_1 = F(y_n, t_n)$$

$$K_2 = F(y_n + \frac{1}{2}\Delta t K_1, t_n + \frac{1}{2}\Delta t)$$

$$K_3 = F(y_n + \frac{1}{2}\Delta t K_2, t_n + \frac{1}{2}\Delta t)$$

$$K_4 = F(y_n + \Delta t K_3, t_n + \Delta t)$$

The user can select either RK2 or RK4 for the simulation. RK4 provides a more accurate result than RK2, but it is slower simulation. Therefore, this function is designed to have both RK2 and RK4, so the user can choose to have a faster or a more accurate simulation for the situation.

4.3 function find_steady_states

This function is used for finding the steady-states of the variables. Steady-state is the solution to the model equations when all the time derivatives are zero, and it is also known as stationary state or equilibrium solution. It is usually used as the initial condition for the variable. The user will need to amend this function if the equations of the model are changed. The procedure of this function for finding the steady-states is:

Step 1: *Reads in Parameters & Constants*
The constants, control-flags and user-defined parameters are recalled from the global at this step. The most significant parameters for the modified Liley model are ΔV^{rest}, λ_{Ach} and λ_i, because these parameters are directly relate to the steady-states of the variables. Equation 4.27 and 4.28 in Section 4.4 define the relationship between these parameters and the steady-states of Q_e and Q_i.

Step 2: *Defines the Steady-States for V_e, V_i, Q_e & Q_i*
`Function steady_states_delV_lambda` is called here to solve for the steady-states with the user-specified input parameters (ΔV_e^{rest}, λ_{Ach} and λ_i) and it will return the value of the steady-states for V_e, V_i, Q_e and Q_i. The user needs to supply this function. This function is currently for the modified Liley model. If the equations change, this code also must be modified. Figure 4.3 presents the steady-states in terms of the equilibrium mean excitatory soma potential as a function of ΔV_e^{rest} and λ_{Ach}. It shows the relationship between the V_e steady-states, ΔV_e^{rest} and λ_{Ach} for the modified Liley model, where the thick green line marks the turning points (the gradient is infinite). If the cross section of V_e and λ_{Ach} with negative ΔV_e^{rest} is viewed, an S-band is found. This means there are three steady-states existing with a defined ΔV_e^{rest} and λ_{Ach}. This S-band can be divided into three branches: upper, middle and bottom branches. The top and bottom branches are usually stable and the middle branch is unstable. As ΔV_e^{rest} becomes more and more positive, this S-band will disappear. The band is not in S shape anymore when ΔV_e^{rest} is positive. This means there is only one steady-state existing in this situation. Therefore, `function steady_states_delV_lambda` might return either one or three steady-states with the user-defined ΔV_e^{rest} and λ_{Ach}. (This function will never return a result of two steady-states even if it is on the connection between upper and middle branches or bottom and middle branches. It will return three steady-states with two of them are the same value due to the way this function had been coded.)

Step 3: *Gets the Steady-States with selected Branch*
The steady-state of the user-defined branch (i.e. top, middle or bottom branch) will be selected from the returned values at this step. However, if there is only

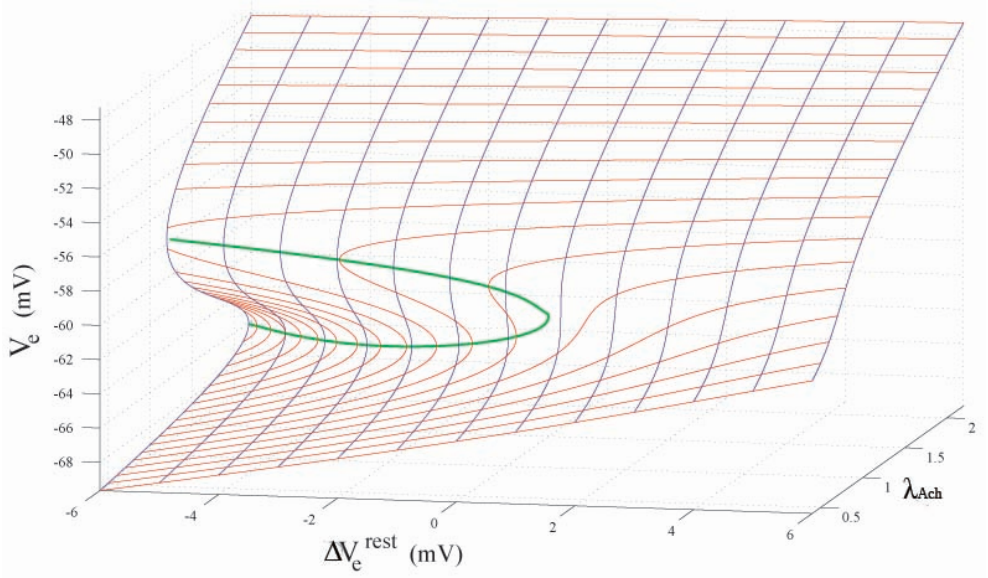


Figure 4.3: *Diagram of Steady-State. Source: Modified from Ref. (15)*
This shows the steady-state of V_e in terms of ΔV_e^{rest} and λ_{Ach}. There is usually one steady-state. However, the green line marks the turning point. An S-band can be observed here if the cross section of V_e and λ_{Ach} is viewed. This results three steady-states in this region.

one branch, then this state will be selected. Since the unstable steady-state is not usually important, so this system does not provide the middle branch for the user to select. Only the upper and bottom branches are available for the user to select. The values in the returned steady-state vector of the variable from `function steady_states_delV_lambda` are ranging from more to less negative. This means the order of the branches for the returned steady-state is: bottom (1^{st}), middle (2^{nd}) and upper (3^{rd}). If the bottom branch is selected, then the first value of the returned steady-state vector will be picked. If the upper branch is selected, it is dangerous to pick the **third** value from the returned steady-state vector, because it might cause an error if there is only one returned steady-state. Therefore, this system uses a MATLAB built-in command `end` to pick the **last** value of the vector for the upper branch. The error is eliminated by this way of coding. If there is only one value in the returned steady-state vector, then both upper and bottom branches return the same value and the setting of the control flag `I.branch` takes no effect for this kind of situation. The selected values of V_e, V_i, Q_e and Q_i will be returned at this step.

Step 4: *Loads-up the equilibrium values for other variables*
Since `function steady_states_delV_lambda` for the modified Liley model only returns the equilibrium values for the variables Q_e, Q_i, V_e and V_i, so it is necessary to load them up to other variables. There are 16 variables after the equations of Liley model spread out and four of them are Q_e, Q_i, V_e and V_i. Table 4.1 shows the

Table 4.1: Output Order of the Variables

Index	**Varaible**
1	V_e
2	V_i
3	Φ_{ee}
4	$Xee(=\frac{d\Phi_{ee}}{dt})$
5	Φ_{ei}
6	$Xei(=\frac{d\Phi_{ei}}{dt})$
7	Φ_{ie}
8	$Xie(=\frac{d\Phi_{ie}}{dt})$
9	Φ_{ii}
10	$Xii(=\frac{d\Phi_{ii}}{dt})$
11	ϕ_{ee}
12	$Yee(=\frac{d\phi_{ee}}{dt})$
13	ϕ_{ei}
14	$Yei(=\frac{d\phi_{ei}}{dt})$
(won't be output by `function find_steady_states`)	Q_e
(won't be output by `function find_steady_states`)	Q_i

variables in the modified Liley model.

Since the variables Q_e and Q_i can be can solved analytically, because they are the functions of V_e and V_i (equations 3.9 and 3.10 in Chapter 3), so they do not need to be processed by numerical integration. Therefore, they are not going to be returned to the main function. This leaves only 14 variables now for processing.

The equilibrium values / equations needed to be recalculated for the other variables. The original Liley's Equations are listed in Chapter 3. The equations 3.1 — 3.8 are the second-order differential equations in time. A set of fourteen coupled first-order differential equations can be derived based on these by splitting Equation 3.1 — 3.8 (3):

$$\frac{dV_e}{dt} = \frac{1}{\tau_e}(V_e^{rest} + \Delta V_e^{rest} - V_e + \lambda_{Ach}\rho_e\psi_{ee}\Phi_{ee} + \lambda_i\rho_i\psi_{ie}\Phi_{ie}) \tag{4.1}$$

$$\frac{dV_i}{dt} = \frac{1}{\tau_i}(V_i^{rest} - V_i + \lambda_{Ach}\rho_e\psi_{ei}\Phi_{ei} + \lambda_i\rho_i\psi_{ii}\Phi_{ii}) \tag{4.2}$$

$$\frac{d\Phi_{ee}}{dt} = X_{ee} \tag{4.3}$$

$$\frac{dX_{ee}}{dt} = -2\gamma_{ee}X_{ee} - \gamma_{ee}^2\Phi_{ee} + \gamma_{ee}^2(N_{ee}^{\alpha}\phi_{ee} + N_{ee}^{\beta}Q_e + \phi_{ee}^{sc}) \tag{4.4}$$

$$\frac{d\Phi_{ei}}{dt} = X_{ei} \tag{4.5}$$

$$\frac{dX_{ei}}{dt} = -2\gamma_{ei}X_{ei} - \gamma_{ei}^2\Phi_{ei} + \gamma_{ei}^2(N_{ei}^{\alpha}\phi_{ei} + N_{ei}^{\beta}Q_e + \phi_{ei}^{sc}) \tag{4.6}$$

$$\frac{d\Phi_{ie}}{dt} = X_{ie} \tag{4.7}$$

$$\frac{dX_{ie}}{dt} = -2\gamma_{ie}X_{ie} - \gamma_{ie}^2\Phi_{ie} + \gamma_{ie}^2(N_{ie}^{\beta}Q_i + \phi_{ie}^{sc}) \tag{4.8}$$

$$\frac{d\Phi_{ii}}{dt} = X_{ii} \tag{4.9}$$

$$\frac{dX_{ii}}{dt} = -2\gamma_{ii}X_{ii} - \gamma_{ii}^2\Phi_{ii} + \gamma_{ii}^2(N_{ii}^{\beta}Q_i + \phi_{ii}^{sc}) \tag{4.10}$$

$$\frac{d\phi_{ee}}{dt} = Y_{ee} \tag{4.11}$$

$$\frac{dY_{ee}}{dt} = -2\upsilon\Lambda_{ee}Y_{ee} - \upsilon^2(\Lambda_{ee}^2 - \nabla^2)\phi_{ee} + \upsilon^2\Lambda_{ee}^2Q_e \tag{4.12}$$

$$\frac{d\phi_{ei}}{dt} = Y_{ei} \tag{4.13}$$

$$\frac{dY_{ei}}{dt} = -2\upsilon\Lambda_{ei}Y_{ei} - \upsilon^2(\Lambda_{ei}^2 - \nabla^2)\phi_{ei} + \upsilon^2\Lambda_{ei}^2Q_i \tag{4.14}$$

All the differential terms must be zero for the steady-states, so Equation 1.3 to 1.14 now become:

$$\phi_{ee}^{eqm} = Q_e^{eqm} \tag{4.15}$$

$$\phi_{ei}^{eqm} = Q_e^{eqm} \tag{4.16}$$

$$\Phi_{ee}^{eqm} = N_{ee}^{\alpha}\phi_{ee}^{eqm} + N_{ee}^{\beta}Q_e^{eqm} + \phi_{ee}^{sc} \tag{4.17}$$

$$\Phi_{ie}^{eqm} = N_{ie}^{\beta}Q_i^{eqm} + \phi_{ie}^{sc} \tag{4.18}$$

$$\Phi_{ei}^{eqm} = N_{ei}^{\alpha}\phi_{ei}^{eqm} + N_{ei}^{\beta}Q_e^{eqm} + \phi_{ei}^{sc} \tag{4.19}$$

$$\Phi_{ii}^{eqm} = N_{ii}^{\beta}Q_i^{eqm} + \phi_{ii}^{sc} \tag{4.20}$$

$$X_{ee}^{eqm} = 0 \tag{4.21}$$

$$X_{ei}^{eqm} = 0 \tag{4.22}$$

$$X_{ie}^{eqm} = 0 \tag{4.23}$$

$$X_{ii}^{eqm} = 0 \tag{4.24}$$

$$Y_{ee}^{eqm} = 0 \tag{4.25}$$

$$Y_{ei}^{eqm} = 0 \tag{4.26}$$

Because the variables X_{ee}, X_{ei}, X_{ie}, X_{ii}, Y_{ee} and Y_{ei} are differential terms originally, so they are all zero for the steady-states. The other variables can now be defined by

the equilibrium equations above.

Step 5: *Tests the Stability*
The stability of the steady-state is going to be defined after the steady-state of the variables is derived. `Function Steady_States_stability_14x14` will be called here and the code shown in this work evaluates the stability of the stationary states of the modified Liley model. Again, if the user wishes to use a new model of the cortex, he must supply a new function here. Since this function is used for testing the stability of the steady-states, so it is triggered by `function find_steady_states` here after the steady-states are defined, not by other functions. `Function Steady_States_stability_14x14` then returns the eigenvalues, largest eigenvalue and oscillation frequency of the variables. The eigenvalues, largest eigenvalue and oscillation frequency will be discussed in detail in Section 4.5.

Step 6: *Returns Steady-States to the main function*
The cortex is assumed to be an `Nspace`×`Nspace` grid plane in the simulation. The variables will be returned to the main function as an `Nspace`×`Nspace` matrix with the same steady-state value on every point of the grid.

4.4 function steady_states_delV_lambda

This function was written by Alistair Steyn-Ross and it is designed to solve for the steady-states with the user-specified input parameters (ΔV_e^{rest}, λ_{Ach} and λ_i) for the modified Liley model.
The main inputs to this function are: ΔV_e^{rest}, λ_{Ach} and λ_i. And the function will calculate and return the steady-states for the variables: V_e, V_i, Q_e and Q_i.
The basic idea for finding the steady-states is by guessing and testing. For example, to find the equilibrium solution for Q_e:

1. *Guess a range of values for Q_e*

2. *Process these guessed Q_e values with the input parameters (ΔV_e^{rest}, λ_{Ach} and λ_i) by Equation 4.27 and 4.28*

$$Q_i = \frac{1}{N_{ie}^{\beta}}[\frac{V_e - V_e^{rest} - \Delta V_e^{rest} - \lambda_{Ach}\rho_e\psi_{ee}(N_{ee}^{\alpha\beta}Q_e + \phi_{ee}^{sc})}{\rho_i\lambda_i\psi_{ie}} - \phi_{ie}^{sc}] \tag{4.27}$$

$$Q_e = \frac{1}{N_{ei}^{\alpha\beta}}[\frac{V_i - V_i^{rest} - \rho_i\lambda_i\psi_{ii}(N_{ii}^{\beta}Q_i + \phi_{ii}^{sc})}{\lambda_{Ach}\rho_e\psi_{ei}} - \phi_{ei}^{sc}] \tag{4.28}$$

3. *Find the guessed Q_e value which matches with (or close to) the analytical value*
This guessed value is the equilibrium solution for the variable. An equivalent procedure applies to Q_i as well. The steady-states for variables V_e and V_i can

be found by the equations (Equation 3.9 and 3.10 shown in Section 3.3.1) if Q_e and Q_i are defined.

Once the steady-states for the variables V_e, V_i, Q_e and Q_i are all defined, then they will be returned to `function find_steady_states`.
Equation 4.27, 4.28, 3.9 and 3.10 are based on Liley model. These equations must be modified if the model is changed.

4.5 `function Steady_States_stability_14x14`

This is the function for storing the partial derivative equations of the variables, and calculating the eigenvalues and the oscillation frequency of the steady-states. It provides a more detailed stability analysis for the steady-state.

4.5.1 Eigenvalue

The eigenvalue is used to define the stability, and it usually contains two parts:

Real part This defines the strength of damping.
E.g. $\lambda = -5 \pm 3j$ is more strongly damped than $\lambda = -2 \pm 20j$

Imaginary part This defines the angular frequency of the oscillation for damping.
E.g. if $\lambda = 3 \pm 7j$, then the angular frequency $\omega = 7rad \cdot s^{-1}$

$\lambda = a \pm jb$ is equivalent to a perturbation decaying by $e^{(a \pm jb)t} = e^{at}e^{jbt}$. This gives $e^{at} \times \cos(bt)$ in the real time, where b here is $2\pi f$, where f is the oscillation frequency. e^{at} is exponential decay if a is negative (Figure 4.4).

Therefore, there are usually three kinds of damping can be observed for the steady-state (Figure 4.5):

Strong damping The eigenvalues are negative. The settling time for the signal is very short. The signal is stable in this case.

Weak damping The eigenvalues form a complex conjugate pair whose real parts are negative. The signal is oscillating and dying away slowly, so the settling time is longer than the **strong damping** case and some oscillation can be observed.

Zero damping The eigenvalues are pure imaginary. The signal keeps oscillating and will not die away.

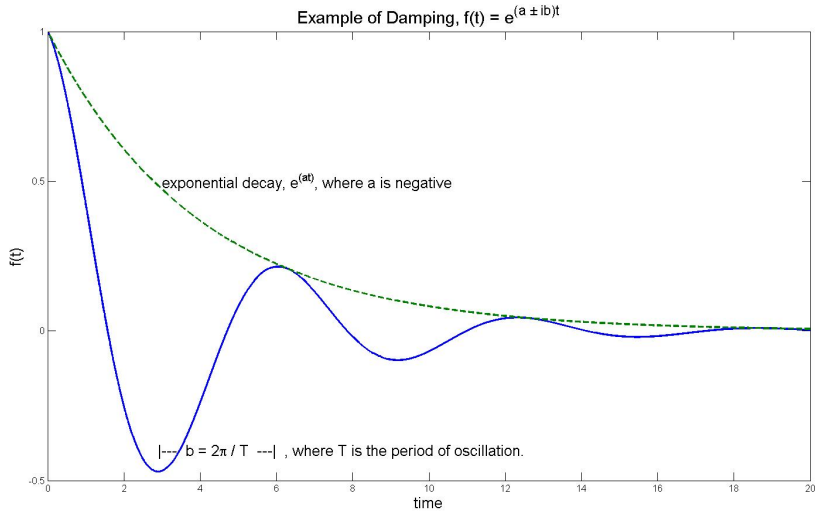


Figure 4.4: Damping definition

This figure illustrates the concept of damping. The blue line is the signal and the green dash line shows the decayed curve of the signal.

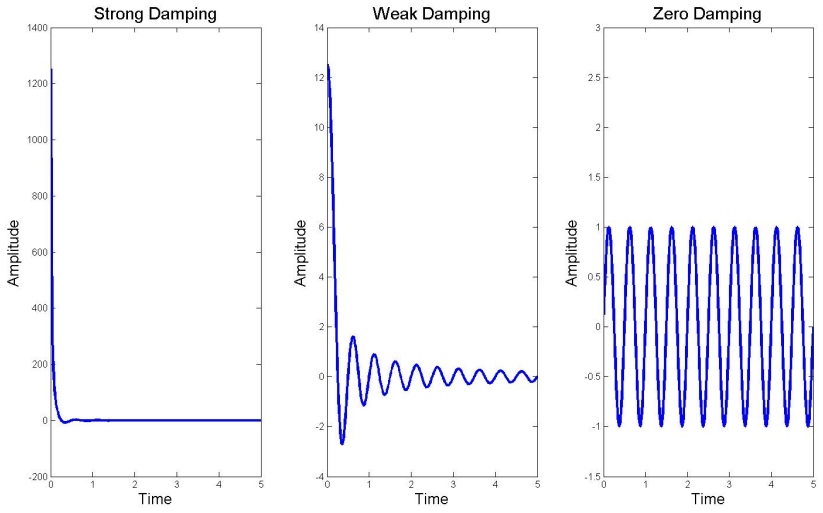


Figure 4.5: Example of Damping

This figure shows three forms of damping. The oscillation amplitude is decreased dramatically for ***strong damping*** *and it is decreased slowly for* ***weak damping****. There is no decay in signal oscillation amplitude for* ***zero damping****. The value displayed on the axes is insignificant here.*

4.5.2 Procedure

The procedure of this function is:

Step 1: *Reads in the parameters and equilibriums*
This function reads in the parameters value, constants, and also the equilibrium of the variables.

Step 2: *Defines the Jacobian matrix*
To define the Jacobian matrix, the user must input the partial derivatives of the model's ordinary equations. For example, if the linearized equations of the model are in the form $\frac{dx}{dt} = f(x,t)$ and the equilibrium is the vector x_0, then the partial derivative can be defined by differentiating the linearized equations by the variables. The partial derivative will be in the form $(\frac{\partial f_i}{\partial x_j})_{x_0}$, where x_j is the j^{th} component of the vector x. If there are 14 variables, then there will be 14×14 partial derivatives. The definition of the partial derivatives must be modified for changing the model.

Since there are 14 variables processed by the system for the modified Liley model, so we have a number of 14^2 partial derivatives here. These partial derivatives are put into a matrix for calculating the eigenvalues, and this 14×14 matrix of partial derivatives is the Jacobian matrix. The equations of the partial derivatives are list below:

$$\frac{\partial \frac{dV_e}{dt}}{\partial V_e} = \frac{1}{\tau_e}(-1 - \frac{\rho_e \lambda_{Ach} \Phi_{ee}}{V_e^{rev} - V_e^{rest}} - \frac{\rho_i \lambda_i \Phi_{ie}}{V_i^{rev} - V_e^{rest}}) \tag{4.29}$$

$$\frac{\partial \frac{dV_e}{dt}}{\partial \Phi_{ee}} = \frac{\rho_e \lambda_{Ach} \psi_{ee}}{\tau_e} \tag{4.30}$$

$$\frac{\partial \frac{dV_e}{dt}}{\partial \Phi_{ie}} = \frac{\rho_i \lambda_i \psi_{ie}}{\tau_e} \tag{4.31}$$

$$\frac{\partial \frac{dV_i}{dt}}{\partial V_i} = \frac{1}{\tau_i}(-1 - \frac{\rho_e \lambda_{Ach} \Phi_{ei}}{V_e^{rev} - V_i^{rest}} - \frac{\rho_i \lambda_i \Phi_{ii}}{V_i^{rev} - V_i^{rest}}) \tag{4.32}$$

$$\frac{\partial \frac{dV_i}{dt}}{\partial \Phi_{ei}} = \frac{\rho_e \lambda_{Ach} \psi_{ei}}{\tau_i} \tag{4.33}$$

$$\frac{\partial \frac{dV_i}{dt}}{\partial \Phi_{ii}} = \frac{\rho_i \lambda_i \psi_{ii}}{\tau_i} \tag{4.34}$$

$$\frac{\partial \frac{d\Phi_{ee}}{dt}}{\partial X_{ee}} = 1 \tag{4.35}$$

$$\frac{\partial \frac{dX_{ee}}{dt}}{\partial X_{ee}} = -2\gamma_{ee} \tag{4.36}$$

$$\frac{\partial \frac{dX_{ee}}{dt}}{\partial \Phi_{ee}} = -\gamma_{ee}^2 \tag{4.37}$$

$$\frac{\partial \frac{dX_{ee}}{dt}}{\partial \phi_{ee}} = \gamma_{ee}^2 N_{ee}^{\alpha} \tag{4.38}$$

$$\frac{\partial \frac{dX_{ee}}{dt}}{\partial V_e} = \gamma_{ee}^2 N_{ee}^{\beta} \frac{\partial Q_e}{\partial V_e} \tag{4.39}$$

$$\frac{\partial \frac{d\Phi_{ei}}{dt}}{\partial W_{ei}} = 1 \tag{4.40}$$

$$\frac{\partial \frac{dW_{ei}}{dt}}{\partial W_{ei}} = -2\gamma_{ei} \tag{4.41}$$

$$\frac{\partial \frac{dW_{ei}}{dt}}{\partial \Phi_{ei}} = -\gamma_{ei}^2 \tag{4.42}$$

$$\frac{\partial \frac{dW_{ei}}{dt}}{\partial \phi_{ei}} = \gamma_{ei}^2 N_{ei}^{\alpha} \tag{4.43}$$

$$\frac{\partial \frac{dW_{ei}}{dt}}{\partial V_e} = \gamma_{ei}^2 N_{ei}^{\beta} \frac{\partial Q_e}{\partial V_e} \tag{4.44}$$

$$\frac{\partial \frac{d\Phi_{ie}}{dt}}{\partial Y_{ie}} = 1 \tag{4.45}$$

$$\frac{\partial \frac{dY_{ie}}{dt}}{\partial Y_{ie}} = -2\gamma_{ie} \tag{4.46}$$

$$\frac{\partial \frac{dY_{ie}}{dt}}{\partial \Phi_{ie}} = -\gamma_{ie}^2 \tag{4.47}$$

$$\frac{\partial \frac{dY_{ie}}{dt}}{\partial V_i} = \gamma_{ie}^2 N_{ie}^{\beta} \frac{\partial Q_i}{\partial V_i} \tag{4.48}$$

$$\frac{\partial \frac{d\Phi_i i}{dt}}{\partial Z_{ii}} = 1 \tag{4.49}$$

$$\frac{\partial \frac{dZ_{ii}}{dt}}{\partial Z_{ii}} = -2\gamma_{ii} \tag{4.50}$$

$$\frac{\partial \frac{dZ_{ii}}{dt}}{\partial \Phi_i i} = -\gamma_{ii}^2 \tag{4.51}$$

$$\frac{\partial \frac{dZ_{ii}}{dt}}{\partial V_i} = \gamma_{ii}^2 N_{ii}^{\beta} \frac{\partial Q_i}{\partial V_i} \tag{4.52}$$

$$\frac{\partial \frac{d\phi_{ee}}{dt}}{\partial M_{ee}} = 1 \tag{4.53}$$

$$\frac{\partial \frac{dM_{ee}}{dt}}{\partial M_{ee}} = -2\upsilon\Lambda_{ee} \tag{4.54}$$

$$\frac{\partial \frac{dM_{ee}}{dt}}{\partial \phi_{ee}} = -\upsilon^2\Lambda_{ee}^2 - \upsilon^2\nabla^2 \tag{4.55}$$

$$\frac{\partial \frac{dM_{ee}}{dt}}{\partial V_e} = \upsilon^2 \Lambda_{ee}^2 \frac{\partial Q_e}{\partial V_e} \tag{4.56}$$

$$\frac{\partial \frac{d\phi_{ei}}{dt}}{\partial N_{ei}} = 1 \tag{4.57}$$

$$\frac{\partial \frac{dN_{ei}}{dt}}{\partial N_{ei}} = -2\upsilon\Lambda_{ei} \tag{4.58}$$

$$\frac{\partial \frac{dN_{ei}}{dt}}{\partial \phi_{ei}} = -\upsilon^2 \Lambda_{ei}^2 - \upsilon^2 \nabla^2 \tag{4.59}$$

$$\frac{\partial \frac{dN_{ei}}{dt}}{\partial V_e} = \upsilon^2 \Lambda_{ei}^2 \frac{\partial Q_e}{\partial V_e} \tag{4.60}$$

(The equations of other partial derivatives not shown here are zero)

These equations are based on Liley model and they must be modified for changing the model.

Step 3: *Outputs the Eigenvalues, Largest Eigenvalues and Oscillation Frequency*
The eigenvalues will be calculated after the Jacobian matrix is defined. Because the Jacobian matrix is a 14×14 matrix, so the returned eigenvalues is a vector with 14 values in it and some of the values are the complex conjugate pair. The least negative value in the eigenvalues vector is significant and should be further processed or checked by user, because it shows how unstable the system is, so it will be derived from the eigenvalues vector. If this derived value (largest eigenvalue) is a complex conjugate pair, then the imaginary part is the angular frequency of the oscillation and this will also be returned for further analysis by the user. The eigenvalues, largest eigenvalue and oscillation frequency will be further discussed in Chapter 6.

4.5.3 Testing Mode

This function has a testing mode in it. The purpose of this is for user to examine the output, to ensure that this it is working properly. This function needs the steady-states before the testing mode starts, so the testing mode of this function is triggered by `function find_steady_states`. The testing parameters are set in `function find_steady_states`, and it works out the steady-states for the testing cases. Then these testing parameters and steady-states are passed to the stability function to calculate the eigenvalues. The returned eigenvalues should be as the user expected if the stability function is working properly.

4.6 `function first_order_derivative`

The equations of the model needed to be linearized to the form of first order derivative equations, due to the requirement by the system for numerical integration.

This function is for storing the first order derivative equation of the variables. It is used by function Runge_Kutta_Integration for numerical integration. It has the following procedures:

Step1: *Read in the previous state of the variables.*
The vector with the value of the variables in the previous round will be readed in. However, the main function will pass the initial steady-states from function find_steady_states through the RK function to this function if it is the first round of the simulation.

Step 2: *Read in the constants, parameters and noise*
These needed to be readed in every time when this function is recalled, because imitated white noise is different every time step and some parameters might be updated to a new value.

Step 3: *Defines the first order derivatives of the variables*
The model used for the design is the modified Liley model. However, the model can be changed by the user if necessary and this function is one of the functions that must be modified for changing model. There are 14 variables being processed for numerical integration, so there are 14 equations stored in this function for the variables. The equations of Liley model were listed in the Section 3.3.1. There are 14 first order derivative equations can be derived by linearizing the original Liley's equations. They were also listed in Section 4.3.

Step 4: *Return the first order derivatives of the variables*
The derivatives will be put into a vector for return. The order of the variables in the vector is consistent to the input vector.

4.7 function noise

This is the function for imitating noise. The soma signal of the neuron always has noise in it, and part of which comes from the subcortex. The noise is important to the brain; experimentally a subject becomes unconscious if there is no noise in the neural signal (20). Therefore, the noise is significant and it must be imitated for the simulation.
The basic idea is to generate random numbers from a probability distribution corresponding to a number of subcortical firings. This function is a short function with a procedure of three steps:

Step 1: *Read in the user-input parameters and the average firing rates (unit: s^{-1}) per neuron.*

The random numbers are going to be distributed around the mean value of the firing rates.

Step 2: *Defines the mean number of noise firings in time Δt for an area $\Delta x \cdot \Delta y$*
The mean value is required for generating random numbers from a probability distribution, and it can be defined by:

$$\lambda_\phi = \frac{N_m \phi^{sc} \Delta t}{\Delta x \Delta y}$$

where λ_ϕ is is the mean number of noise firings in time Δt for an area $\Delta x \cdot \Delta y$, $\Delta x \cdot \Delta y$ is the area covered by one grid point, ϕ^{sc} is the mean noise rate per macrocolumn, and N_m is number of macrocolumns in the cortex.

Step 3: *Generates random numbers from the probability distribution*
There are two ways to generate random numbers in this function. One is to pick up the random numbers from a Poisson distribution and another one is to pick up from a Normal distribution. The noise imitated by Poisson is better than by the way of Normal distribution, because it more closely follows what happens in practice when one firing event is uncorrelated to another. However, MATLAB generates the random numbers from a Poisson distribution more slowly than from a Normal distribution. If the mean value is large, then the results from both Poisson and Normal distributions are similar to each other. Therefore, both ways of generating noise have been put into this function and the user can select either one.
(The standard deviation for the normal distribution here is the square root of the mean value derived at Step 2)

Step 4: *Pass the imitated noise terms to global*
These noise terms **R** (R_{ee}, R_{ei}, R_{ie} and R_{ii}) will be called from global by first-order derivative function for numerical integration, where R_{ee}, R_{ei}, R_{ie} and R_{ii} are the random matrixes for ϕ^{sc}.

4.8 `function find_trajec`

This is the function for defining the trajectory. The value of the parameters might change with time if there is a trajectory (e.g. γ_i, λ_i, λ_{Ach}, ΔV_e^{rest}, etc...). This is the reason why other functions need to update these parameters for every time step during the simulation. The user can add the trajectory to this function to define how the parameters change with time and the function can contain several trajectories in it. The user can select the required trajectory by simply setting the control-flag `I.trajectory` in the user-input file.
Here is one example for a trajectory: application of the anaesthetic drug **isoflurane**.

Isoflurane (`I.trajectory = 1`)

This is the trajectory for the anaesthesia. It is to simulate the situation if the drug has been administered to the patient. This trajectory controls the parameters γ_i (the inhibitory rate constant) and λ_i (a scaling of the IPSP). The equations below show how these two parameters change with time.

Ratio of changing for λ_i:

$$\lambda_i = -3.7335x^3 + 2.1785x^2 + 4.4188x + 1.0125$$

If $x < 0.6$, then the ratio of changing for γ_i:

$$\gamma_i = -1.9017x^3 + 4.4384x^2 - 3.228x + 0.9878$$

If $x \geq 0.6$, then the ratio of changing for γ_i:

$$\gamma_i = -1.9017x^3 + 4.4384x^2 - 3.228x + 0.9878$$

(Where $x = drug_{max}\frac{t_{current}}{t_{total}}$ and $drug_{max} = 1.2$)

The equations above were derived from the statistics of the clinical data (by Banks and Pearce) (21). The values derived by the equations are the ratios of changing for γ_i and λ_i. These ratios multiply by γ_i & λ_i of the previous round to get new values for γ_i & λ_i for present round of the simulation. A new equation can be added into this function if there is a new drug applies.

4.9 `function perturbation`

This is the function for applying a step change in the cortical signal during the simulation. Since the effect of a perturbation can show on an EEG (e.g. as a k-complex (22)), so this function is designed to simulate this kind of situation. This function is used to see what would happen to the neurons if there is a sudden change in the cortical signal at one point in space. However, this kind of perturbation is still complicated, so there is no definition or equation for it and its parameters are still user-defined. The user can define the characteristics of the perturbation (e.g. time scale, magnitude, spatial localization).

4.10 `function Input`

This function is for storing the user defined inputs. It includes:

- Parameters required by the model

Table 4.2: The standard parameters required by the modified Liley model

Parameter	Representation in MATLAB	Description
ΔV_e^{rest}	`del_Vrest`	Effect of altering extrasynaptic ion channels
λ_{Ach}	`lambda_Ach`	Effect of muscarinic neuromodulation
λ_i	`lambda_i`	Multiplier on inhibitory synaptic weight
γ_e	`gamma_e`	Excitatory synaptic rate constant
γ_i	`gamma_i`	Inhibitory synaptic rate constant
$\Lambda_{ee,ei}$	`lamda_ee`	Characteristic inverse length-scale for connections
—	`nu`	Relative amount of noise
υ	`v`	Mean axonal conduction speed

- Parameters required by the system
- Control-flags
- File name of the output data
- Axes limit, tick and label for the plot

This input function is different to other input functions. The other input functions only needed to be changed if the model changes, or if there is a new thing to be added to the system; otherwise other input functions are kept unchanged while using this system. This function stores the parameters required by each function for the simulation, so the user needs to check and modify the values here every time before starting the simulation. Therefore, this is the function that the user will change the most while using this system.

At the time this report was being written, the model is based on modified Liley model, where the equations of the model were discussed in Section 3.3.1. Table 4.2 lists the parameters required by modified Liley model stored in this function. The user can change the value for the parameters above for simulation. However, the code for the parameters above must be modified for changing the model.

There are some key parameters affect the simulation directly in this function. The parameter `Nspace` is the number of points on the coordinate for x and y axis. This defines the density of the points on the cortex plane grid and this affects the resolution of the simulation. Other parameters are `deltat` and `tend`, they are the time step-size and the total time steps. The total time for the simulation can be defined by multiplying both together (unit in second).

Since most control-flags here are alternative with two statuses: enable and disable; and the other input functions are designed to use if-else statement with the control-flag to select the status, so the control-flags are set by using TRUE or FALSE in these if-else statements. TRUE is readed as 1 and FALSE is readed as 0 in MATLAB. The user can input 1 or 0 instead of TRUE or FALSE as well for the control-flags in the user input file. The logic for this way of design is to prevent the user inputs any other value for the control flags, because any other value is true in else statement.

4.11 function data_output

This function is designed to do three jobs:

1. Outputs the data at the present time step of the simulation to global (collection of data for saving)
2. Generates a plot of a user-selected variable
3. Records the frame for a movie

However, the three functions above are designed to be implemented if the user wishes it. The user can do any one, or all, or none of them during the simulation. The reason for the three functions above are designed to let the user to define is because they all slow the simulation speed.
The first one above slows the simulation speed the most after examining the system performance. It was designed to record every variable initially, but it is modified to save the user-selected variables. This improves the performance of the simulation, but the user might need to take more time to think what they need for analysis. This function can trigger the steady-state function for recording the stability data (stability function is triggered by steady-state function) if the control-flag of the stability testing is activated. The reason for putting it into this function is because the data of stability testing is useful for the analysis if there is a trajectory. Another thing that needs to be mentioned here is that the saved data might not contain every time step of the simulation, because the change between each time step might be small. Then recording for every time step is unnecessary here and it slows the simulation speed and leads to memory storage problems. Therefore, it is designed to record the data once every fixed interval of time steps and this interval (number of time steps) can be defined by the user.

4.12 function save_output

Since `function data_output` passes the data to global, so this function saves the data from the global. The data from `function data_output` is only one single time-step length each time, so it passes to global for collection. Then this function is called to save the data after the data of every time-step is collected. At the time of this report was being written, the saved data contains variables, stability data, parameters and also a movie of the simulation. The user can change the parameters being saved by modifying this function. The filename of the saved data can be changed in user-input file. The saved movie will be in AVI format. The reason for converting the MATLAB movie to AVI format is because that the movie in AVI format can be opened by Microsoft windows media player, and the MATLAB movie can only be opened under MATLAB window. Since MS Windows is a popular

operating system, so the movie saved in AVI format is more convenient. However, the data will only be saved if the control flag for saving is activated, otherwise no data will be saved (for example, this is for the case if the user just wants to test the simulation).

Chapter 5

Detailed System Design — Output Processing

After the simulation is done and the data is saved, the next stage is to process the saved simulated data. There are two functions, which are designed to process the output data from the simulation. One is to process for the power spectrum of the cortical signal and another one is to display the time variation of a user-selected parameter. A user is of course able to write further functions of his own if he requires them.

5.1 `function Power_Spectrum`

This function is for processing the cortical signal data of the simulation and generates the plot of power spectrum. The **power** here for the power spectrum is the square of the cortical signal strength. This function defines the power and plots it against temporal and spatial frequency, where the temporal frequency defines how frequently the signal fluctuates in time and the spatial frequency defines how frequently the signal fluctuates in space. For example, if the temporal frequency is high, this means the difference of signal strength between each time step is large. If the spatial frequency is high, this means the difference between each point in space is large. The user can see the variation of the cortical signal in space and time from the plots and further analyze the result.

5.1.1 Numerical Implementation for Fourier Transform (3)

The data of the simulated cortical signal must be Fourier transformed before processing for the power spectrum. If V(t) is our simulated continous cortical signal, then it can be Fourier transformed to get the corresponding spectrum:

$$\tilde{V}(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} V(t) e^{-j\omega t} dt \tag{5.1}$$

with the Inverse Fourier Transform being:

$$V(t) = \int_{-\infty}^{\infty} \tilde{V}(\omega) e^{j\omega t} d\omega \tag{5.2}$$

where $\tilde{V}(\omega)$ are the scaled continuous Fourier components of V(t). And the corresponding discrete expressions of Equation 5.1 and Equation 5.2 are:

$$\tilde{V}(\omega) = \frac{\Delta t}{2\pi} \sum_{t=0}^{T} V(t) e^{-j\omega t} \tag{5.3}$$

$$V(t) = \frac{2\pi}{T} \sum_{\omega=0}^{\infty} \tilde{V}(\omega) e^{j\omega t} \tag{5.4}$$

where T is the total time period for a simulated data series.

The power spectra can be obtained by Fourier transforming the computer-simulated data and extract the temporal and spatial components. Generally **power** is discussed in terms of a continuous spectrum, because the theoretical power spectrum is a continuous function. However, the computer-simulated data for the cortical signal is discrete. Therefore, we must interpret the transformed data as a discrete sampling of a continous function. The normalization of the discretely sampled spectrum must be done such that the total mean power in the system equals the continous integral over all frequencies of the power spectrum. For example, for the temporal component, it requires that:

$$\int_{\omega=0}^{\infty} |\tilde{V}(\omega)|^2 d\omega = \frac{1}{T} \lim_{T\to\infty} \int_{t=0}^{T} |V(t)|^2 dt \tag{5.5}$$

However, the equation above must be replaced by discrete summations for a discrete Fourier transform, so the equation becomes:

$$\sum_{k=0}^{M-1} |\tilde{V}_k|^2 (\frac{2\pi}{T}) = \frac{1}{T} \sum_{i=0}^{M-1} |V_i|^2 \Delta t \tag{5.6}$$

Where V_i (note this is different to the inhibitory soma potential V_i, **i** here represents the time $t = i\Delta t$) is the simulated cortical signal, which represents the discrete form of $V(t)$. $\tilde{V}_k$ is the Fourier component of V_i, which is the discrete form of $\tilde{V}(\omega)$, and the relationship between k and ω is $\omega = \frac{2\pi k}{T}$. The term $\frac{2\pi}{T}$ in the left hand side is the increment in frequency ($\Delta\omega = \frac{2\pi}{T}$). M is the number of time intervals in the period T ($M = \frac{T}{\Delta t}$). This equation can be further simplified to:

$$\sum_{k=0}^{M-1} |\tilde{V}_k|^2 = \frac{\Delta t}{2\pi} \sum_{i=0}^{M-1} |V_i|^2 \tag{5.7}$$

The power spectra can be obtained by taking the modulus squared of $\tilde{V}_k$.
It is important to realize that a discrete fourier transform (e.g. `fft`) in Matlab, being simply a numerical algorithm, does not generate correctly scaled fourier components for this application. The relationship between $\tilde{V}_k$ and V_i tells us how to scale the fourier components V_k generated by the discrete fourier transform. If each V_k is scaled by the following factor, the relationship of Equation 5.7 will be obeyed:

$$ScaleRatio = \frac{\Delta t}{2\pi} \sum_{i=0}^{M-1} |V_i|^2 / \sum_{k=0}^{M-1} |\tilde{V}_k|^2 \tag{5.8}$$

After multiplying by this factor the fourier components V_k correctly represent a discrete sampling of the continuous $\tilde{V}(\omega)$ spectrum.

An equivalent procedure applies to the spatial components. The equation for equivalence in power is:

$$\sum_{q_x=0}^{N_x-1} \sum_{q_y=0}^{N_y-1} |\tilde{V}_{q_x q_y}|^2 = \frac{\Delta x \Delta y}{4\pi^2} \sum_{i=0}^{N_x-1} \sum_{j=0}^{N_y-1} |V_{ij}|^2 \tag{5.9}$$

where N_x and N_y are the number of points in the x and y coordinates of the cortex grid plane and Δx & Δy are the spacing increment in the x and y direction. V_{ij} is the value of the function at the grid point with co-ordinates (i,j) and $V_{q_x q_y}$ is the Fourier component at the reciprocal lattice point (q_x, q_y).
Once again, it needs to be scaled to get the correct power spectrum. The scale ratio can be obtained by rearranging the equation above:

$$ScaleRatio = \frac{\Delta x \Delta y}{4\pi^2} \sum_{i=0}^{N_x-1} \sum_{j=0}^{N_y-1} |V_{ij}|^2 / \sum_{q_x=0}^{N_x-1} \sum_{q_y=0}^{N_y-1} |\tilde{V}_{q_x q_y}|^2 \tag{5.10}$$

And the corresponding temporal and spatial frequencies are: Temporal frequency, $\omega = \frac{2\pi N_t}{T}$ where T is the period and N_t is the increment of discrete time element from the start to the end of the simulation. Spatial Frequency, $q = \frac{2\pi N_q}{lcortex}$ Where `lcortex` is the length of the cortex and N_q is the increment of discrete point on the cortex grid plane.

5.1.2 Programming Implementation

Figure 5.1 shows the procedure of this function.

1. This function loads the data saved by the main function, and also the user-defined time-length for settling time and division period (Figure 5.2 & Figure 5.3).

2. The signal may exhibit transient fluctuations early in the simulation (i.e. not

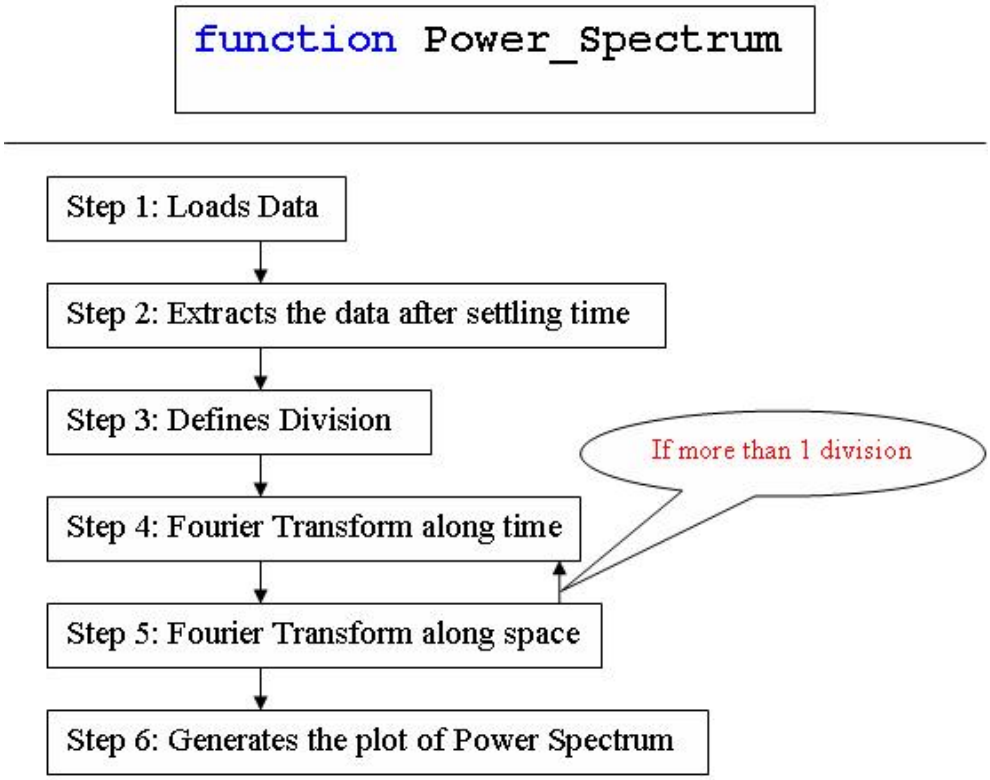


Figure 5.1: *Flowchart of function Power_Spectrum*
This loops Step 4 and Step 5 to process the data division by division if there are more than one division, where the time length for the division period is defined by the user.

have **settled**). Therefore, the user supplies a time period known as the **settling time** in which the data is not worth processing and analysis and the data within the settling time is going to be taken out from the loaded data. However, since the settling time length is user-defined, so an if-else statement has been used here in the code to prevent the user to set any settling time longer than the data time length of the simulation. If the input settling time is longer than the simulation time of the saved data, then it will cause an error and stop the processing.

3. The user might want to divide the data into several pieces by a user-defined time-length and process each division separately. This function works out the total number of divisions for processing at this step. However, since the division period is user-defined, where the division period is the user-defined time-length for division, so an if-else statement has been used here to prevent the user selecting any division period longer than the data time-length of the simulation. If the input division period is longer than the simulation time of

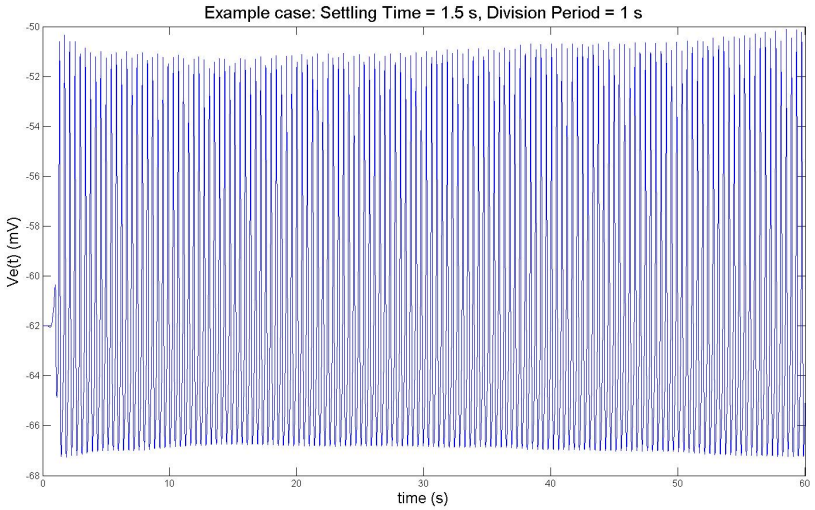


Figure 5.2: *Example for showing the concept of settling time and division period*
This is the example case that the cortical signal supposes to oscillate from the start, but the cortical signal is not oscillating at the start of the simulation, so the data of this none-oscillation period is the settling period and it is not significant to be further processed.

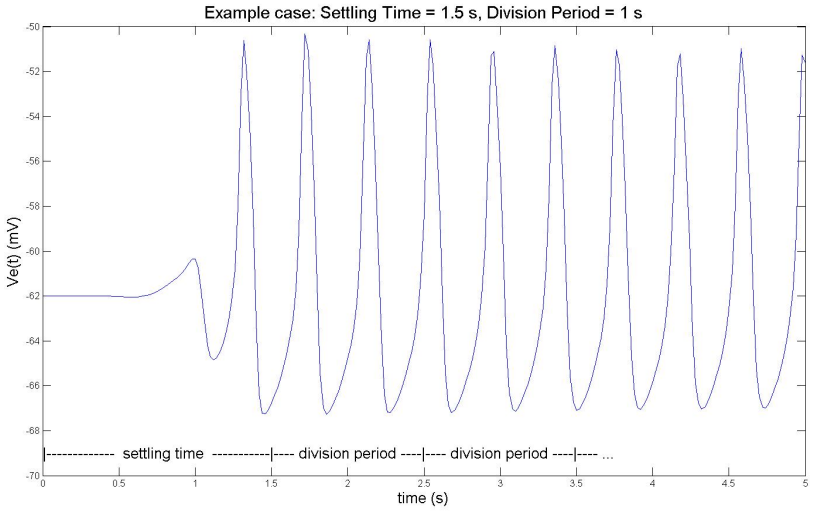


Figure 5.3: *Zoom in of Figure 5.2*
This zooms in the previous figure and shows the oscillation at the start of the simulation. This indicates the time position of the settling time and division period in the time axis of a computer-simulated data.

the saved data, then the function will process the data with full time length as only one division.
If there is more than one division, then the time length of the saved data might not be divisible by the division period time length. This results a remainder with last few time sets of data. The time length of this remainder is shorter than the division period and will not be processed.

4. The Fourier transform for the power spectra starts here and it is processed division by division. At this step, the temporal Fourier component of the power spectrum from the simulation is extracted. The data of simulated cortical signal, V_i, is in discrete time before transform. The MATLAB function `fft` is used here to Fourier transform V_i to get the Fourier component V_k, then the power spectra can be obtained by scaling the V_k square. The programming implementation here for defining the mean power is to Fourier transform the data of the simulated cortical signal point by point of the grid on the cortex plane, and then sum up and average the power spectra of each point.

5. At this step, the spatial Fourier component of the power spectrum from the simulation is extracted. The method of Fourier transform for the spatial component is a bit different to the temporal component. For the temporal component, it is done by transforming the time series point by point for each point on the cortex plane, and then averaging the result.
E.g.

```
for x = 1:Nspace
    for y = 1:Nspace
        Vk = fft(V(x, y, t = 1∼tend));
        ...
    end
end
```

where `tend` is the number of time steps of a division and `Vk` is a vector of numbers of a variable.

For the spatial component, the Fourier transform is done line by line in this function (i.e. Fourier transform $V(x = 1, y = 1 \sim N_y, t)$, then Fourier transform $V(x = 2, y = 1 \sim N_y, t)$, and so on...). This is because the Fourier transform in the previous step is a 1-D transform, and here it is a 2-D Fourier transform in the x a spatial component and time. Therefore, the MATLAB built-in function `fft2` is used here. The Fourier component $V_{q_x q_y}$ can be defined by looping and Fourier transforming the discrete cortical signal V_{ij} of

every x-coordinate (i.e. $fft2(V_{ij}(x = 1))$, then $fft2(V_{ij}(x = 2))$, and so on...), then y-coordinate, then summing up and averaging the results to get the mean power.
E.g.

```
for x = 1:Nspace
    V_qxqy = fft2(V(x, y = 1∼Nspace, t = 1∼tend));
    ...
end
for y = 1:Nspace
    V_qxqy = fft2(V(x = 1∼Nspace, y, t = 1∼tend));
    ...
end
```

6. The plot will be generated after the mean power spectrum is defined, so the function will calculate the vectors for the temporal and spatial frequencies. The power spectrum will be plotted for each division (Figure 5.4). However, if there are more than one division, then the power spectrum from each division will be summed up and averaged, so it gives the plot of average power spectrum in the end after all the divisions are processed.

This function also contains testing code. The purpose of this is to ensure that this function is working properly, and then this function can provide an accurate power spectrum. The way of testing is to feed the function with a standard set of value for the parameters, and it should return a result as expected by the user. The data of the simulation from the main function will not be loaded if the testing mode of this function is activated. There are testing cases with constant frequency, changes in temporal frequency, and changes in spatial frequency. They can be used to test the calculation of the frequency, space and power spectrum for this function. The details of the testing cases are discussed in User-Guide (Appendix A).

5.2 `function Variable_Variation`

This function is for processing a user-selected variable. It is for the user to observe the variation of the cortical signal in space and time, and analyze the simulation results easily. The procedure of processing for this function is simple. This function loads the data of parameters and variables, and calculates the corresponding time and space vectors, then generates the plot (Figure 5.5) for the variable. This function generates two plots. One shows the fluctuation along the time axis of the variable for a user-defined x & y coordinates on the cortex plane. Another one is the black & white plot which shows the changing of the variable along the time and one space

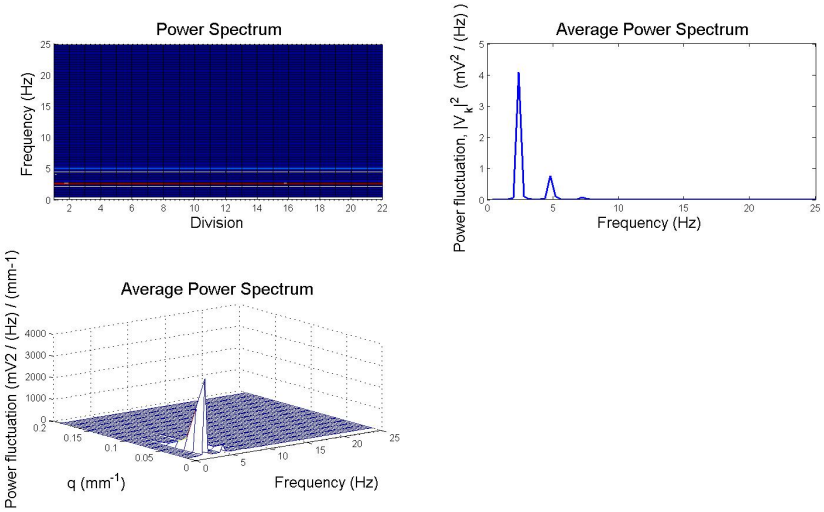


Figure 5.4: *Example for the power spectrum plot*

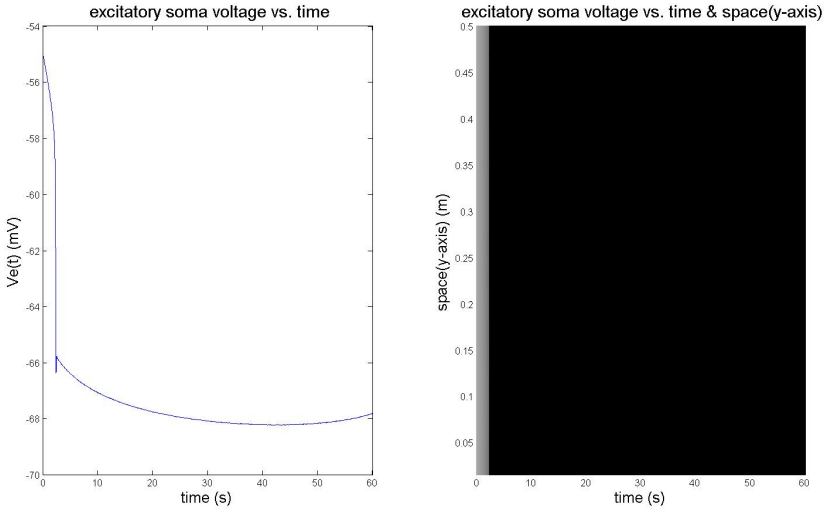


Figure 5.5: *Example for the variable variation plot*

axis (Since the cortex plane is assumed to be square for the simulation, so x & y axes are not significant in direction. Therefore, the space axis of the black & white plot is a fixed y coordinate with ranging in x coordinate in this function.). The detail of these two plots will be discussed in next chapter, but an example of the output is shown here.

Chapter 6

Results and Testing

This system is designed to simulate many kinds of neural situations, so there are many different results that can be generated. They are discussed in this chapter.

6.1 Information of the Initial Conditions

The information of the initial conditions is shown on the MATLAB Command Window when the simulation starts (Figure 6.1). It lists all the corresponding steady-states to the user-input ΔV_e^{rest}, λ_{Ach} and λ_i values. The returned steady-states will be either one or three for each variable. If there are three steady-states returned, then the system will pick up the value of the user-selected branch for the initial condition.

6.2 Stability of the selected steady-state

The system will then display the 14×14 matrix stability test result of the selected steady-state. The meaning of stability is discussed in Section 4.5. Since there are 14 variables of Liley model being processed by the system, so the returned eigenvalues contain 14 values. If all the eigenvalues are negative, then this means the steady-states are stable. However, the larger the largest eigenvalue, the less the stability, even if it is negative. Therefore, the largest eigenvalue is selected from the 14 eigenvalues for considering the overall stability of the system.
Since the variable with the largest eigenvalue has the lowest stability, so it means its oscillation frequency is significant for analysis. The information of 14×14 stability test for the selected steady-state will also be displayed on the Command Window at the start of the simulation (Figure 6.2).

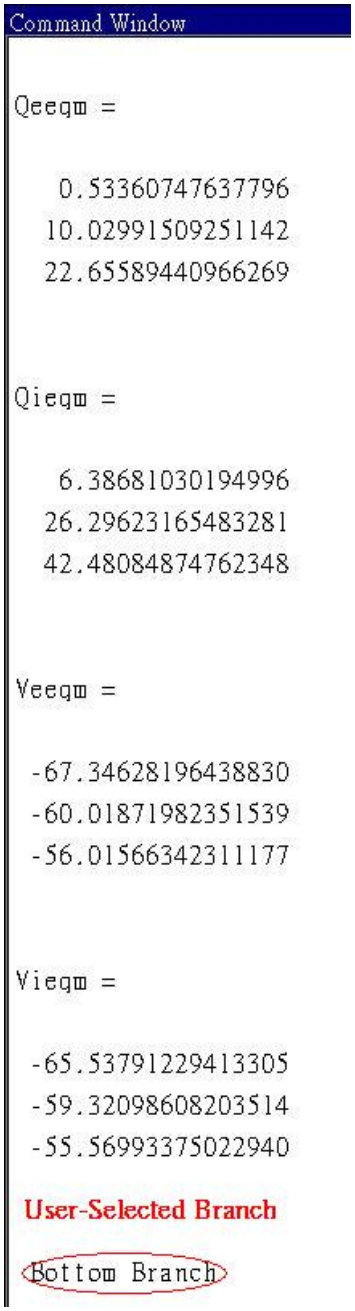


Figure 6.1: *Value of the defined steady-states.*
This shows the steady-states found by the `function steady_states_delV_lambda` *with the user-defined input parameters. The equilibrium value of all the branches (top, middle and bottom branches) for the variables V_e, V_i (unit in mV), Q_e and Q_i (unit in s^{-1}) are displayed in the MATLAB Command Window. And the red circle in the bottom indicates the branch that the system will select the equilibrium value from for processing the steady-state.*

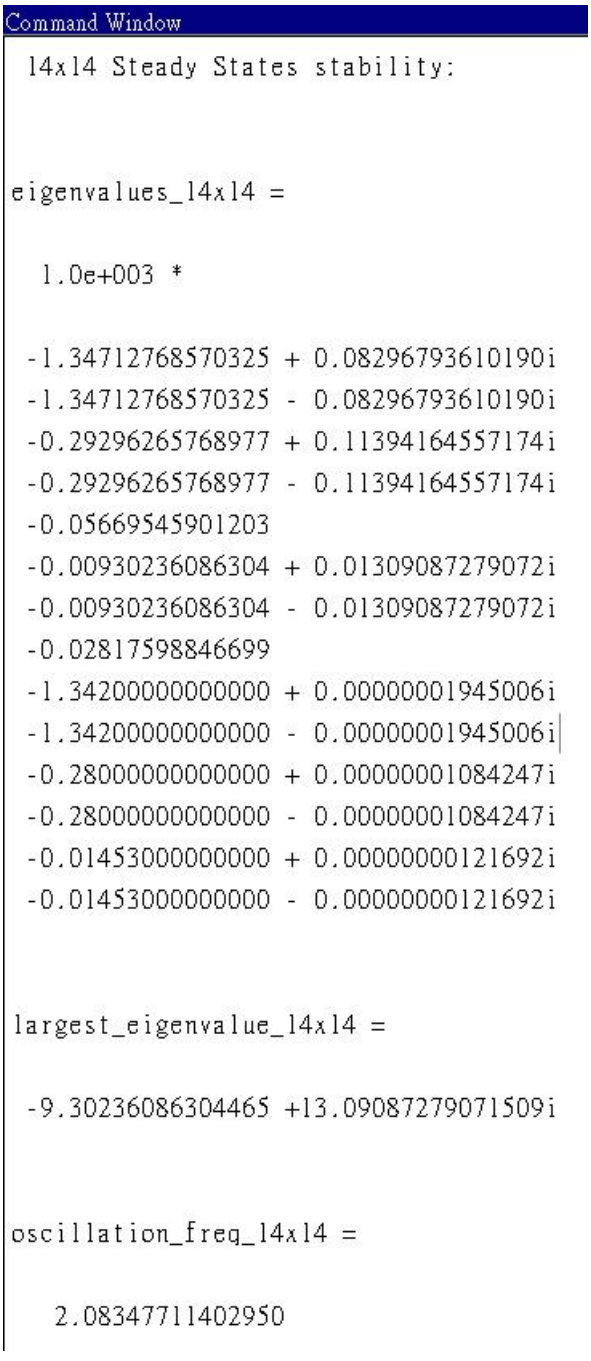


Figure 6.2: *Stability of the selected steady-state where the unit for the eigenvalue is s^{-1} and oscillation frequency is in Hz.*

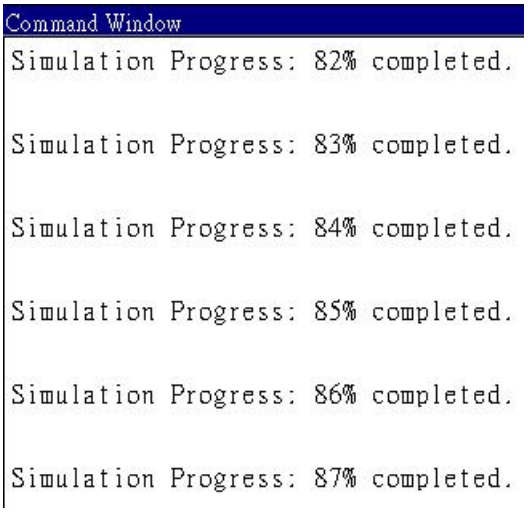


Figure 6.3: *Percentage of completion for the simulation.*

6.3 Simulation

The system then displays the percentage of simulation completion when the looping for the numerical integration starts (Figure 6.3). It is useful for keeping track of the simulation progress.

6.4 Output of the simulation

The system generates two plots while the simulation is running. It shows the changing of the cortical signal on the cortex plane. The size of the grid is determined by the user-defined `Nspace` value. Figure 6.4 is the example of the plots generated by the main function. One is in 2-D view and another one is in 3-D view. The plot of this example case shows a cortex that is at steady-state of about -67mV, which is at bottom branch with only small fluctuations about this. The colour of the grid shows the difference of the variable value between the discrete points on the cortex plane. The red colour represents the larger value and the blue colour represents the smaller value of the variable. The vertical axis in 3-D view shows the exact value of the plotted variable. The user can see the signal fluctuation on the cortex plane from these two plots.

6.5 Trajectory

Trajectory controls parameters that change with time. The user can define how the parameters change with time in the trajectory function. This chapter discusses some examples of the trajectory.

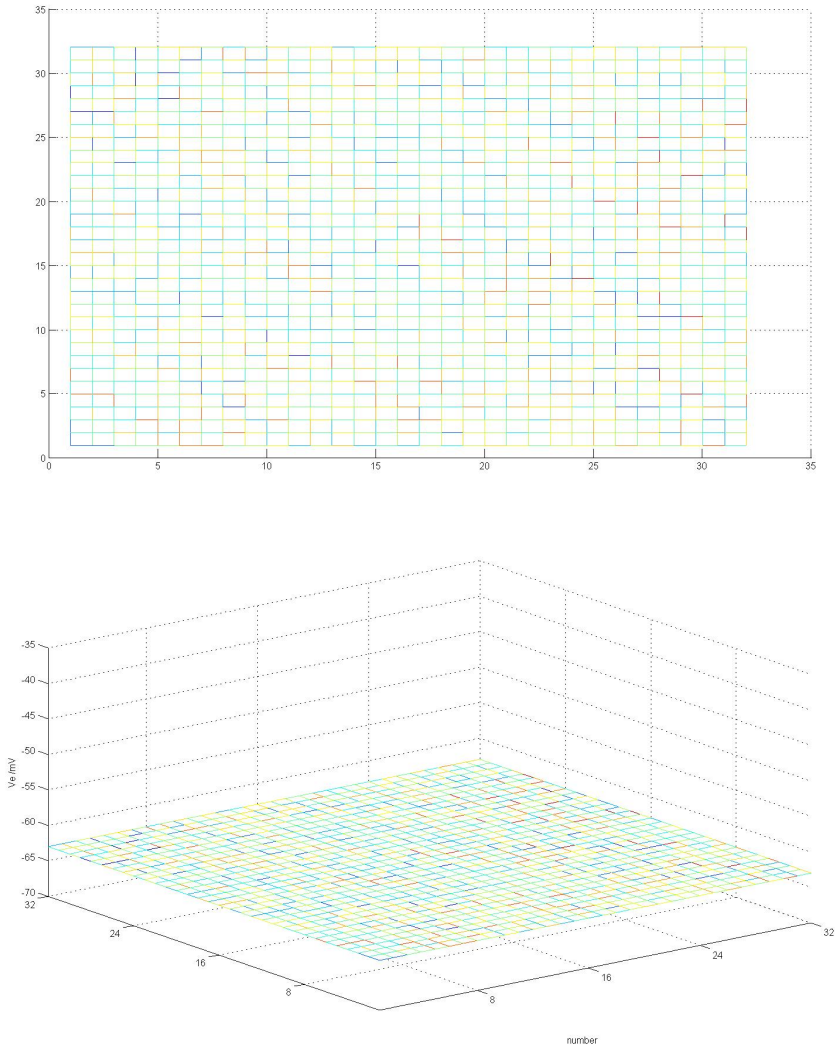


Figure 6.4: *Example of the plots for the simulated cortex plane. The top one is the plot in 2-D view and the bottom one is in 3-D view. The red colour of the grid represents the higher cortical signal strength and the blue colour represent the lower signal strength. The difference of cortical signal strength between each each grid can be clearly observed from the vertical axis of the 3-D view plot.*

6.5.1 No-path: `I.trajectory = 0`

If the trajectory control flag is set to 0, then it will perform a no-path simulation. This means the trajectory is deactivated and there will be no value change with time for each parameter during the simulation.

Unstable initial condition

The example here shows the simulation starts with unstable initial conditions. The value used for the parameters are:

Parameter:	ΔV_e^{rest} (mV)	λ_{Ach}	λ_i	γ_e (s^{-1})	γ_i (s^{-1})
Value:	5	0.5	1	1342	14.53

And the result of 14×14 stability test for the initial condition is:

$$eigenvalues = 1000 \times \begin{pmatrix} -1.35732224221421 + 0.14432926947161i \\ -1.35732224221421 - 0.14432926947161i \\ -0.30761557924783 + 0.18513638455306i \\ -0.30761557924783 - 0.18513638455306i \\ -0.07031337404307 \\ 0.00761241969741 + 0.01291582789753i \\ 0.00761241969741 - 0.01291582789753i \\ -0.03119029085556 \\ -1.34200000000000 + 0.00000003700352i \\ -1.34200000000000 - 0.00000003700352i \\ -0.28000000000000 + 0.00000000959984i \\ -0.28000000000000 - 0.00000000959984i \\ -0.01453000000000 + 0.00000000117832i \\ -0.01453000000000 - 0.00000000117832i \end{pmatrix}$$

There are positive values in the eigenvalues, and the initial condition of this simulation contains unstable states by comparing the values in the table above with Figure 6.5. Therefore, the results of the eigenvalues and the stability plot show that the simulation for this case should be unstable. Both results are in agreement for this case.

Figure 6.6 is the variation for the excitatory soma potential (V_e). The left plot shows that the signal of excitatory soma potential (V_e) is oscillating from the start to the end of the simulation, and the right plot shows that the cortical signal of the whole cortex is oscillating, not just some certain positions. This means the whole cortex is unstable and this simulation result agrees with the 14×14 stability test and stability plot.

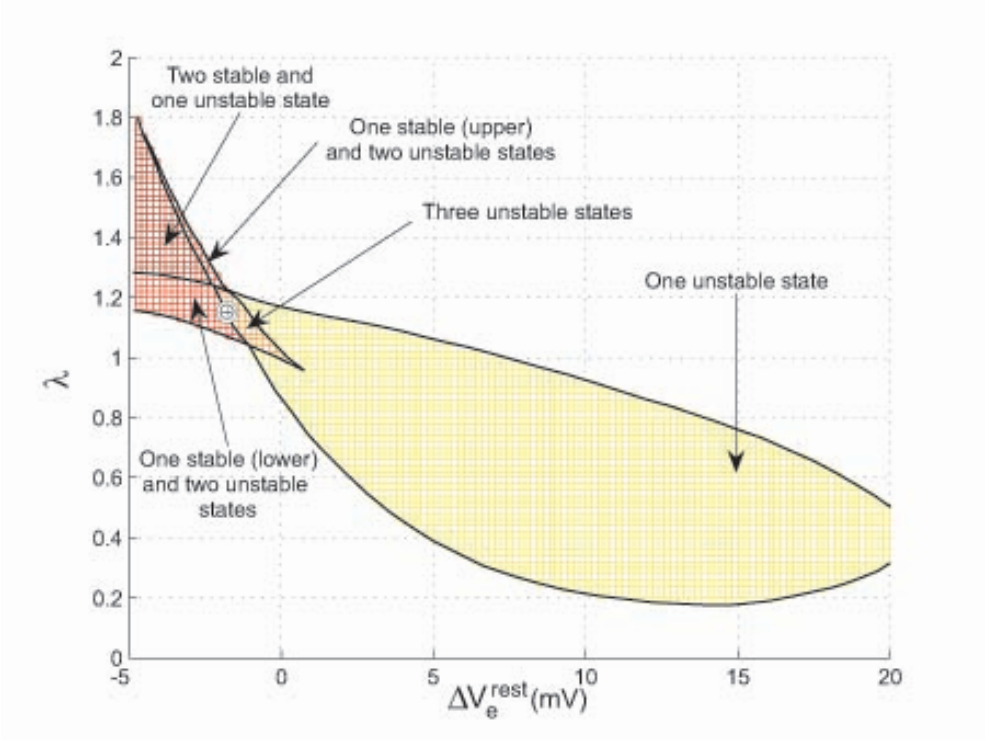


Figure 6.5: Stability Plot of the sleep domain for $\gamma_i = 15s^{-1}$. Source: Ref. (2)
This figure shows the stability when $\gamma_i = 15s^{-1}$. There are five possible situations for the stability: One stable state (unmarked region); one unstable state; three unstable states; one stable and two unstable states, and two stable and one unstable states (the midbranch is always unstable). The point $\oplus$ (ΔV_e^{rest} = -1.8 mV, λ = 1.15) contains three unstable states.

Figure 6.7 is Power Spectrum Analysis. The oscillation frequency can be defined from the plots of power spectrum. The graphs of the power spectrum show three peaks, this means the signal is oscillating at three frequencies: 2.4 Hz, 4.8 Hz and 7.2 Hz (the peak at 7.2 Hz might be hard to see from the graph due to small amplitude). The peaks at 4.8 Hz and 7.2 Hz are 2 and 3 times 2.4 Hz and are therefore probabily harmonics of the oscillation. Since that the lower frequency has a higher peak, and the peak at 2.4 Hz is much larger than other two peaks. This means the Ve signal is mixed by three frequencies and basically oscillating at 2.4 Hz. The amplitude of oscillation is ranging between -51 to -67mV, and it is large to the neural signal. Therefore, the V_e signal is unstable as expected (consistent with the prediction of the eigenvalues and the stability plot) with no changing in state (no-path).

Stable initial condition:

The example here shows the simulation starts with stable initial conditions. The value used for the parameters are:

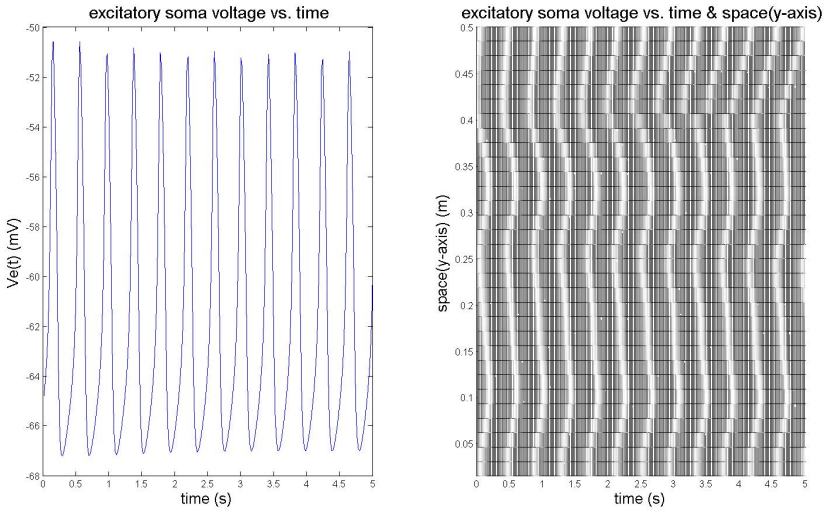


Figure 6.6: *Plot of variation in the cortical signal with unstable initial condition. The left plot shows that the excitatory soma potential V_e of a point (defined by user) is oscillating during the whole simulation. The right plot shows V_e of the whole cortex is oscillating. Both plots show the whole cortex is in unstable condition.*

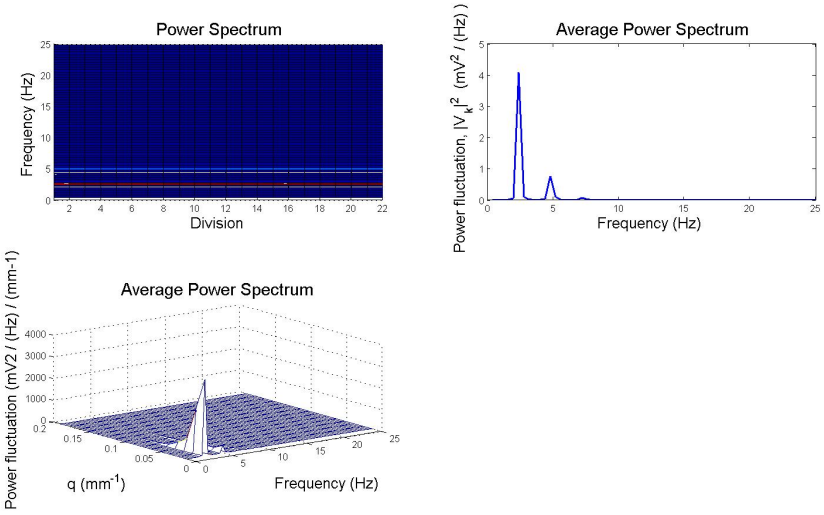


Figure 6.7: *Power Spectrum Analysis.*
There are three frequencies can be observed from these plots: 2.4 Hz, 4.8 Hz, and 7.2 Hz. The peak at 2.4 Hz has the largest amplitude and the peak at 7.2 Hz has the smallest amplitude.

Parameter:	ΔV_e^{rest} (mV)	λ_{Ach}	λ_i	γ_e (s^{-1})	γ_i (s^{-1})
Value:	5	0.3	1	1342	14.53

And the result of 14×14 stability test is:

$$eigenvalues = 1000 \times \begin{pmatrix} -1.34867396763564 + 0.09476036938165i \\ -1.34867396763564 - 0.09476036938165i \\ -0.29571221707072 + 0.12830390710141i \\ -0.29571221707072 - 0.12830390710141i \\ -0.00400205987024 + 0.01574698522650i \\ -0.00400205987024 - 0.01574698522650i \\ -0.05478347324603 \\ -0.03552027332120 \\ -1.34200002028575 \\ -1.34199997971425 \\ -0.28000000000000 + 0.00000000323267i \\ -0.28000000000000 - 0.00000000323267i \\ -0.01453000000000 + 0.00000000159879i \\ -0.01453000000000 - 0.00000000159879i \end{pmatrix}$$

All the eigenvalues above are negative; this means the initial condition is stable. Therefore, the simulation for this case should be stable.

Figure 6.8 is the variation for the excitatory soma potential (V_e). The left plot shows that the peak to peak voltage for the excitatory soma potential is about 0.035mV. This fluctuation is very small and it is caused by noise, not a limit-cycle oscillation in signal. And the plot on the right shows there is no state-change for any discrete point on the cortex plane. The whole cortex system is as stable as expected.

Figure 6.9 is the Power Spectrum Analysis. The spectrogram plot shows red colour for the lower frequencies for every division. And the plots of the power spectrum analysis show that both temporal and spatial frequencies of the largest peak are at 0 Hz with a very small peak amplitude (5.5×10^{-7}). This means there is nearly no signal oscillation for the whole cortex system during the whole simulation. The signal only has noise fluctuation in it. Therefore, the signal of the whole simulation is as stable as expected for this case.

6.5.2 User-path: `I.trajectory = 2`

If the trajectory control-flag is set to 2, then it will perform a user-path simulation. The parameter value is changing during the simulation in a user-defined way. In this case, the value of λ_{Ach} will decrease by 0.25 from the start to the end of the

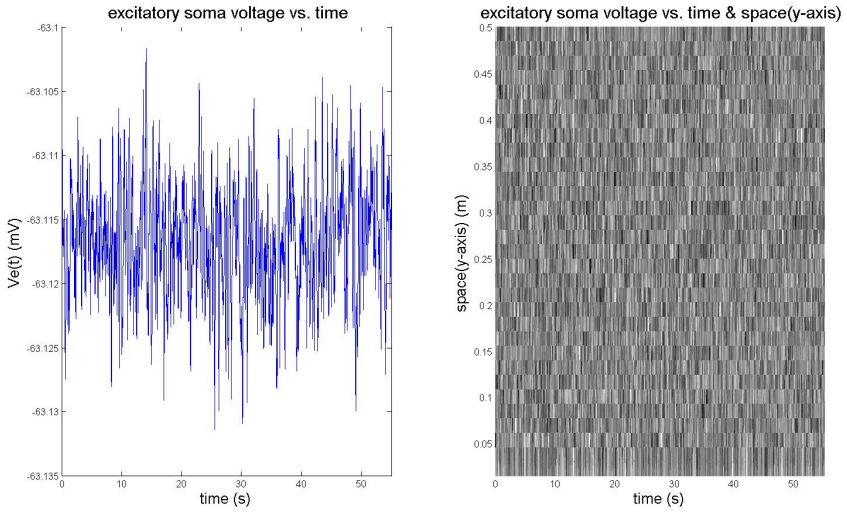


Figure 6.8: Plot of variation in the cortical signal with unstable initial condition. *The left plot shows that V_e is oscillating with a very small peak to peak amplitude and the right plot seems noisy. Both plots show the fluctuation of the cortical signal is caused by noise.*

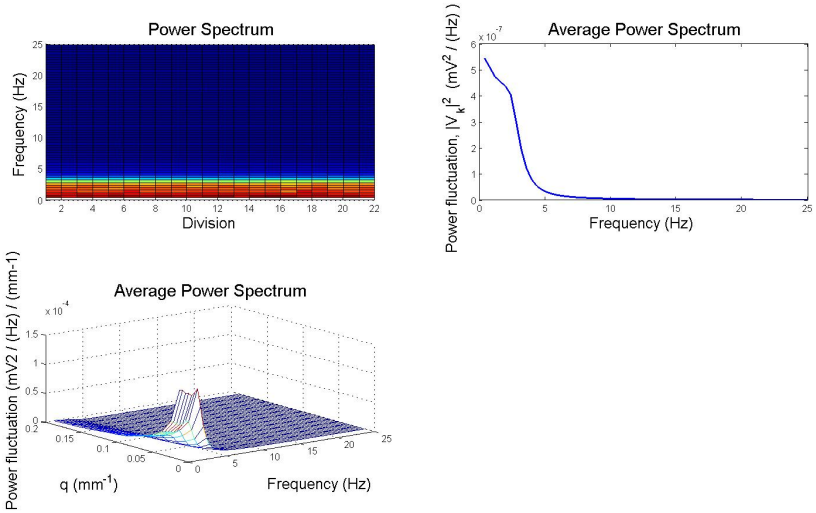


Figure 6.9: *Power Spectrum Analysis.*
These plots show the strength of the cortical signal is very small and it has the largest amplitude when close to 0 Hz. This means the fluctuation of the cortical signal is caused by noise and this is in agreement with Figure 6.8

simulation. Since the starting value for λ_{Ach} is 0.5, so it will be 0.25 in the end. The stability plot shows that this path will pass the boundary between the stable and unstable states (The boundary for this case is when λ_{Ach} is about 0.39).
Stable initial condition:

Parameter:	ΔV_e^{rest} (mV)	λ_{Ach}	λ_i	γ_e (s^{-1})	γ_i (s^{-1})
Value:	5	0.5	1	1342	14.53

The parameters will be input with the values in the table above for this case.

The result of 14×14 stability test for the initial condition ($\lambda_{Ach} = 0.5$) of this case is:

$$eigenvalues = 1000 \times \begin{pmatrix} -1.35732224221421 + 0.14432926947161i \\ -1.35732224221421 - 0.14432926947161i \\ -0.30761557924783 + 0.18513638455306i \\ -0.30761557924783 - 0.18513638455306i \\ -0.07031337404307 \\ 0.00761241969741 + 0.01291582789753i \\ 0.00761241969741 - 0.01291582789753i \\ -0.03119029085556 \\ -1.34200000000000 + 0.00000003700352i \\ -1.34200000000000 - 0.00000003700352i \\ -0.28000000000000 + 0.00000000959984i \\ -0.28000000000000 - 0.00000000959984i \\ -0.01453000000000 + 0.00000000117832i \\ -0.01453000000000 - 0.00000000117832i \end{pmatrix}$$

There are positive values in the eigenvalues. This means that this path is starting from an unstable state. And this is also consistent with the result by comparing the parameters value with the stability plot.
Figure 6.10 is the Path of the trajectory in the stability plot.

Therefore, this path should start with unstable state and end up with stable state for this case.

Figure 6.11 is the variation for the excitatory soma potential (V_e). There is a dramatic change (in the middle of the simulation) can be observed in this figure. The V_e signal was oscillating at the start of the simulation, but its fluctuation is getting smaller and smaller along the simulation time, then goes to steady, as the system moves from an unstable region to a stable region.

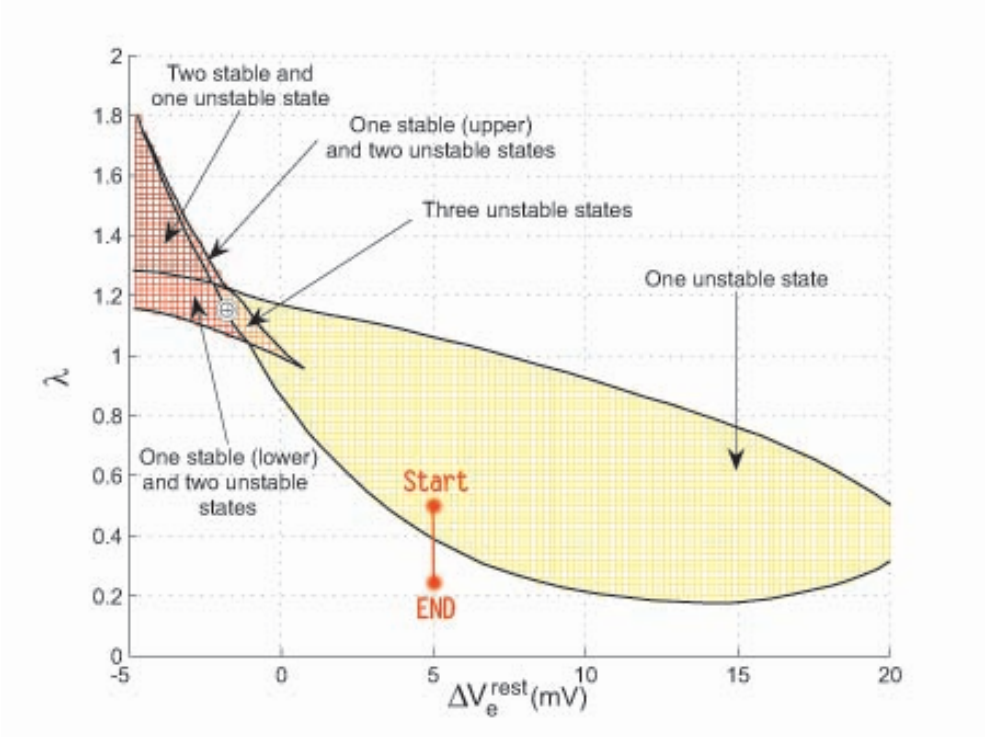


Figure 6.10: *Path of the trajectory in the stability plot.*
This plot shows the path of the trajectory for the case in Section 6.5.2. The trajectory starts at $\lambda = 0.5$ and ends at $\lambda = 0.25$ ($\Delta V_e^{rest} = 5$).

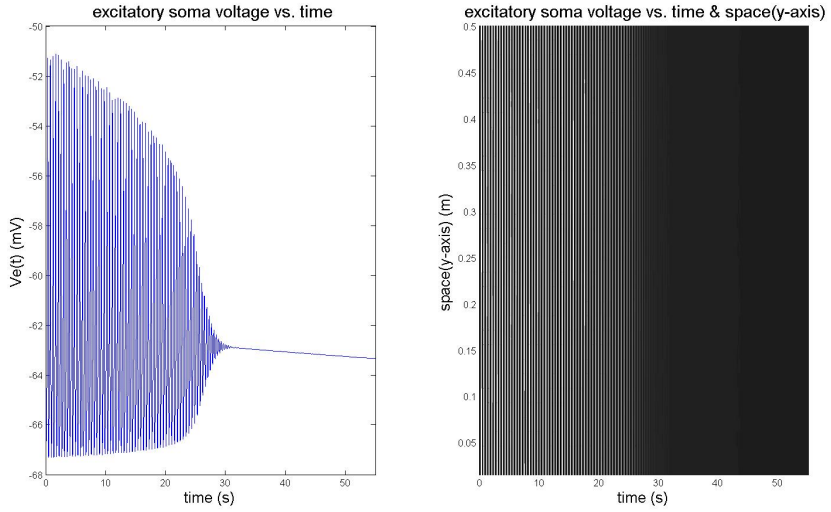


Figure 6.11: *Plot of variation in the cortical signal with user-defined trajectory.*
This figure shows V_e is oscillating at the start, then goes to steady at the half of the simulation. The time of crossing the boundary can be easily observed from these plots.

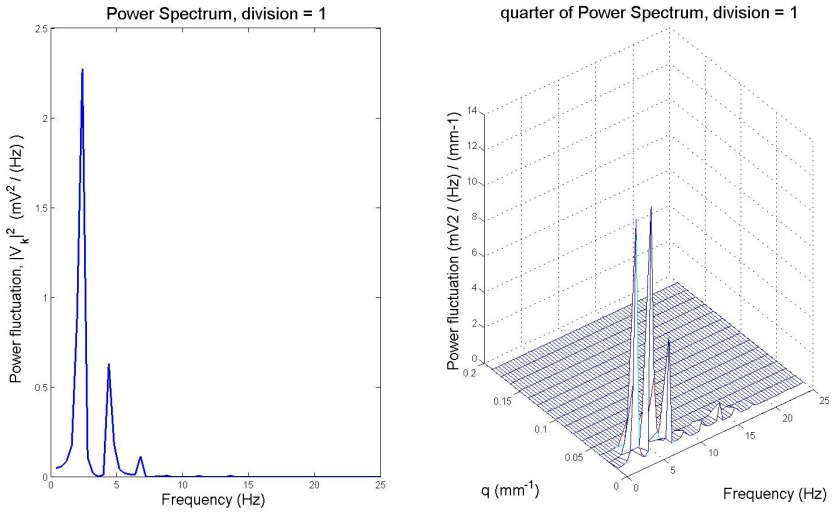


Figure 6.12: Division 1 of the case in Section 6.5.2.
This is the start of the simulation and there are three peaks at different frequencies by looking at the plot above, so the signal combines three frequencies and the peak amplitude is larger at lower frequency. This shows the cortical signal is oscillating and the cortex system is unstable at the start of the simulation.

Power Spectrum Analysis:

Figure 6.12: Division 1
Figure 6.13: Division 11
Figure 6.14: Division 12
Figure 6.15: Division 13
Figure 6.16: Division 14
Figure 6.17: Overall

6.5.3 Isoflurane: `I.trajectory = 1`

The user can input a trajectory for drug to the function if it has equations describing how the parameters change with time. The example drug used in this case for testing the system is **isoflurane**. It controls the parameters γ_i and λ_i to take effect on neurons. The data for the equations of isoflurane comes from Banks and Pearce data (21). Since it is an anaesthetic drug, so it should be able to bring the cortical signal from top to bottom branch. Figure 6.18 shows how the ratios of γ_i and λ_i changes with the simulation time for the drug **isoflurane**. This shows that the γ_i ratio is decreasing exponentially to a constant ratio at 50% of simulation progress. The time scale of the plot is the percentage of the simulation progress, not a certain amount of time, so these two ratios are changing by the way that the plot shows, no matter how long or how short is the simulation time.

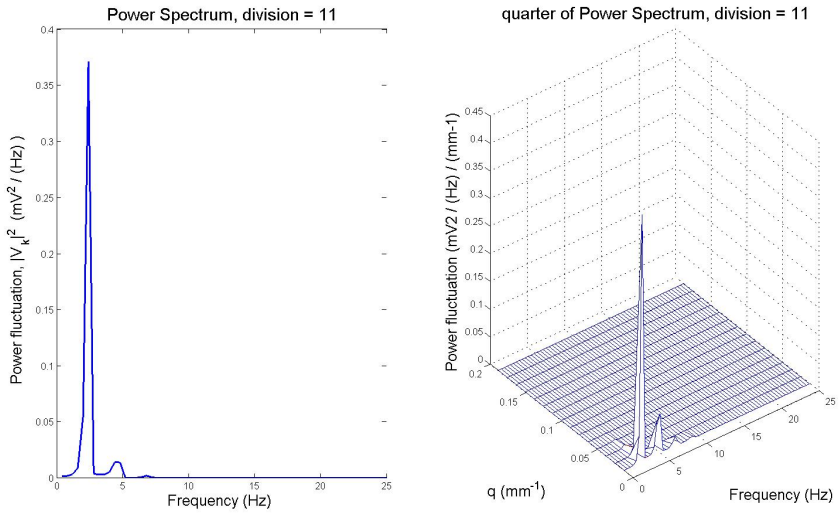


Figure 6.13: *Division 11 of the case in Section 6.5.2.*
This division is about the time just before the dramatic change showed in V_e variation plot above. By comparing this plot with the previous plot, the amplitude for all peaks is getting smaller, and the two peaks at higher frequencies are much smaller than the one at lowest frequency. This shows the cortical signal now is still in the unstable region, but close to the boundary of stable and unstable regions.

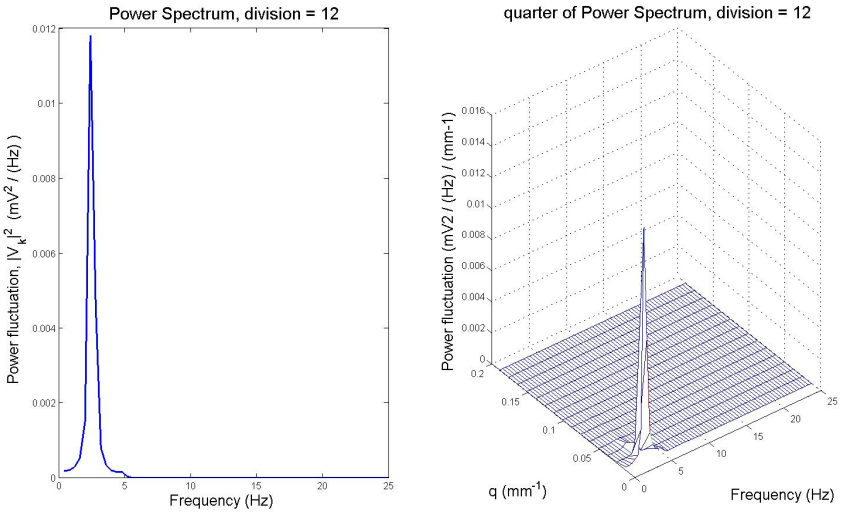


Figure 6.14: *Division 12 of the case in Section 6.5.2.*
The signal in this division only has one frequency in this division. This means the signal now is oscillating at a constant frequency. And the peak amplitude is smaller than the one in previous plot. This shows the cortical signal now is still in the unstable region, but right beside the boundary of stable and unstable regions.

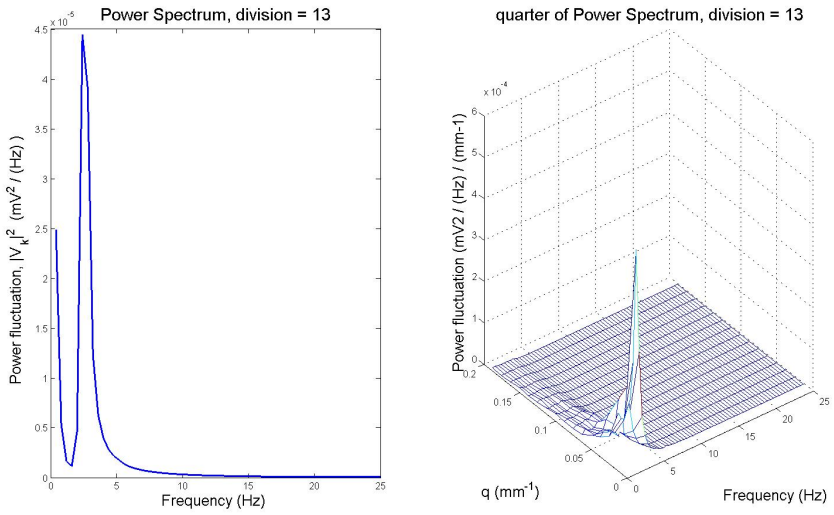


Figure 6.15: *Division 13 of the case in Section 6.5.2.*
The peak amplitude is decreased dramatically in this division by comparing with previous division. This should be the time that the cortical signal crosses the boundary from unstable to stable region.

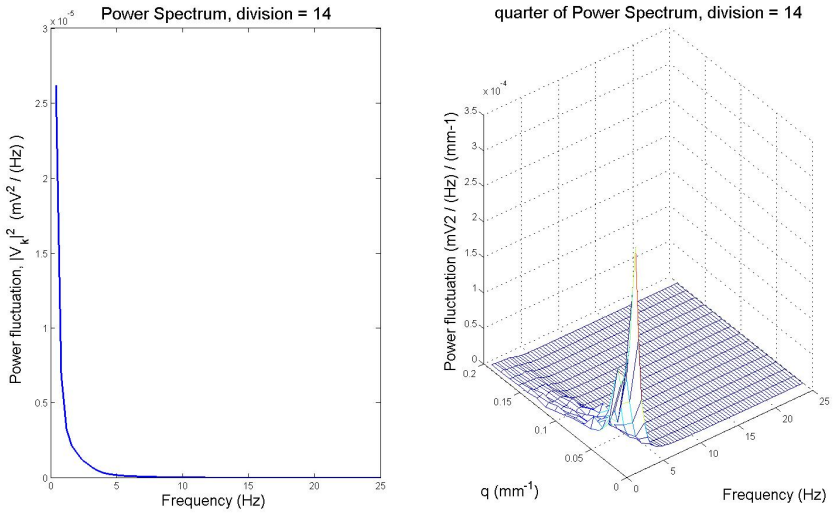


Figure 6.16: *Division 14 of the case in Section 6.5.2.*
By looking at the plot above, the signal is now settled in this division with nearly no oscillation (highest peak at 0 Hz). This shows the cortical signal is now in the stable region.

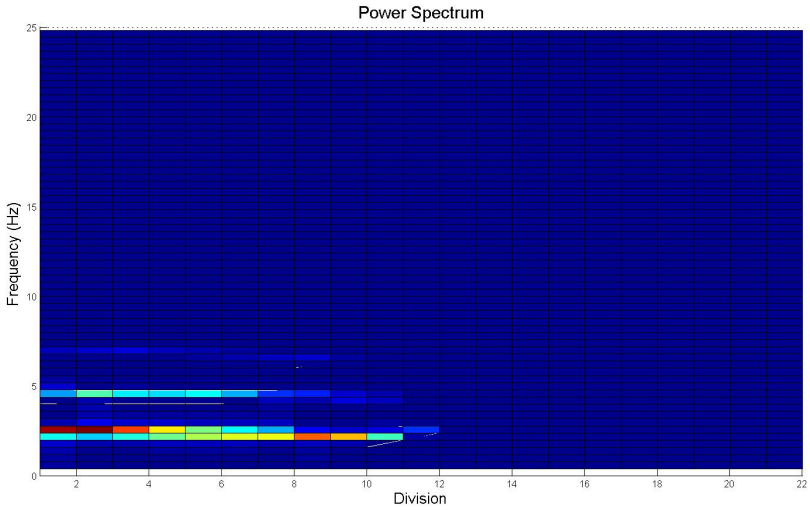


Figure 6.17: *Spectrogram of the user-defined trajectory.*
This shows that the signal is oscillating at the start of the simulation, but the oscillation is getting slower and slower, then finally stopped after it passed the boundary of stable and unstable states. The spectrogram above also shows colour changing for some certain frequencies from division 1 to division 11, and there is no colour changing after division 12. These plots are in agreement with the prediction before the simulation. The system is working as expected for user-path trajectory.

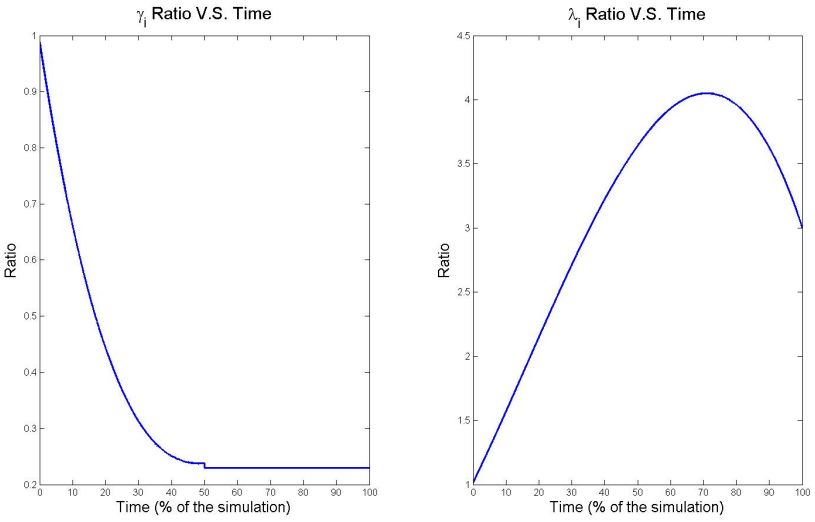


Figure 6.18: *Time variation of γ_i and λ_i ratios.*

For testing the drug path for the trajectory, the values in the table below are input for the parameters for this case:

Parameter:	ΔV_e^{rest} (mV)	λ_{Ach}	λ_i	γ_e (s^{-1})	γ_i (s^{-1})
Value:	-1.8	1.25	1	300	65

The cortical signal starts from top branch of the steady-states for this case. If the drug is working properly, the cortical signal should end up with bottom branch. The stability test result for this case is:

$$eigenvalues = 100 \times \begin{pmatrix} -4.93534321876748 + 1.59163223954284i \\ -4.93534321876748 - 1.59163223954284i \\ -2.02224105845700 + 2.45298948539951i \\ -2.02224105845700 - 2.45298948539951i \\ -0.23171214298867 + 0.44370315970442i \\ -0.23171214298867 - 0.44370315970442i \\ -0.94538111961818 \\ -2.43122418321691 \\ -3.00000000004809 + 0.00000290924339i \\ -3.00000000004809 - 0.00000290924339i \\ -2.79999999995358 + 0.00000346422027i \\ -2.79999999995358 - 0.00000346422027i \\ -0.64999999999999 + 0.00000004613407i \\ -0.64999999999999 - 0.00000004613407i \end{pmatrix}$$

There is no positive value in the eigenvalues vector, so the stability test result shows that the initial condition for this case is stable. The drug should take effect after the simulation start.

Figure 6.19 is the variation for the excitatory soma potential (V_e). The left plot shows that the excitatory soma potential is -55mV at the start, and then goes to about -68mV quickly corresponding to a switch from the top branch to the bottom branch. The right plot shows a short white region on the left and long black region on the right. It means the cortical signal for the whole cortex system goes from top to bottom branch quickly after the simulation start.

Figure 6.20 shows the cortical signal has the largest amplitude at 0 Hz. The power spectrum plot shows some different colours at the bottom left corner of the plot, this shows the cortical signal is oscillating at a very low frequency (nearly 0 Hz) when it is in the top branch, then goes steady when it is in the bottom branch. These can be concluded that the cortical signal is stable during the whole simulation, even when there is a state change. Therefore, the drug isoflurane is working **properly**

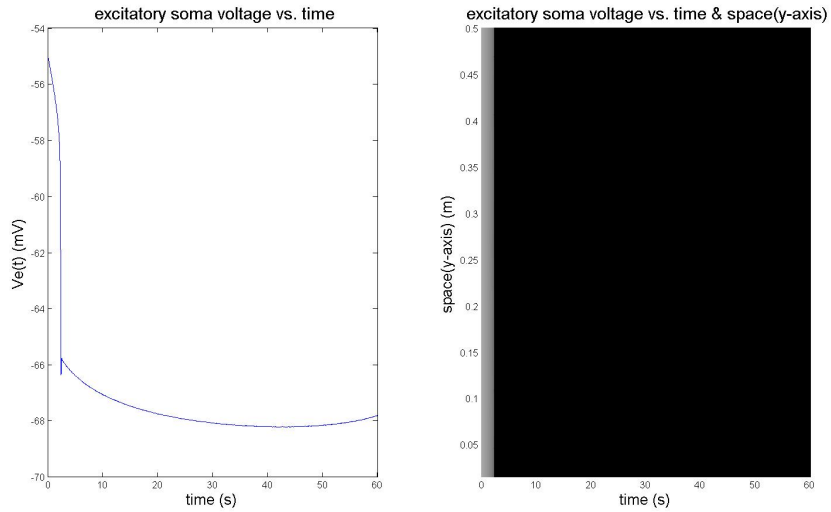


Figure 6.19: *Plot of variation in the cortical signal for isoflurane. This shows V_e is at the top branch at the start, then quickly goes to the bottom branch.*

for this case.

6.6 Testing for Power Spectrum

Wilson et al. had produced a set of power spectra and they are shown in his paper (2). If `function Power_Specturm` in this system is working properly, then it should be able to generate the same plots as above. However, in Wilson et al, a different definition of spatial power was used. M.T. Wilson performed two Fourier transforms in space, so that power is present as power per unit temporal frequency per unit spatial frequency squared ($mV^2Hz^{-1}m^2$). In my interpretation, I perform a single Fourier transform in space. Therefore, the **power** here is presented as power per unit temporal frequency per unit spatial frequency ($mV^2Hz^{-1}mm$).

6.6.1 $\gamma_i = 65s^{-1}$

The table below shows the parameters value (2) required by this case.

Parameter:	ΔV_e^{rest} (mV)	λ_{Ach}	λ_i	γ_e (s^{-1})	γ_i (s^{-1})
Value:	5	0.5	1	300	65

And Figure 6.21 shows the equivalent power spectrum.

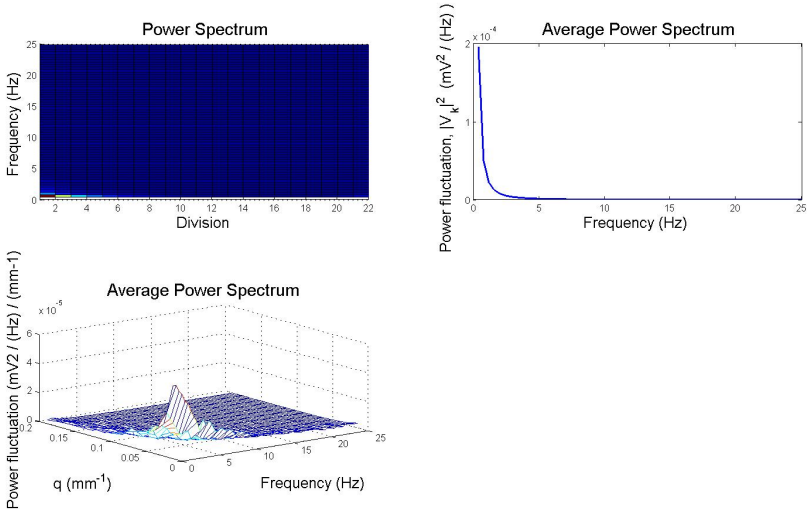


Figure 6.20: *3 in 1 power spectra plot.*
This shows the largest peak is at the frequency close to 0 Hz. This means there is nearly no oscillation and the whole cortex is stable even when there is a state-transition.

6.6.2 $\gamma_i = 53s^{-1}$

The table below shows the parameters value (2) required by this case.

Parameter:	ΔV_e^{rest} (mV)	λ_{Ach}	λ_i	γ_e (s^{-1})	γ_i (s^{-1})
Value:	5	0.5	1	367	53.07

And Figure 6.22 shows the equivalent power spectrum.

6.6.3 $\gamma_i = 46s^{-1}$

The table below shows the parameters value (2) required by this case.

Parameter:	ΔV_e^{rest} (mV)	λ_{Ach}	λ_i	γ_e (s^{-1})	γ_i (s^{-1})
Value:	5	0.5	1	424.3	46.96

And Figure 6.23 shows the equivalent power spectrum.

6.6.4 Overall

If the three power spectra simulated by this system's function are compared with the plots of the paper, they are similar to them. They all have a peak with the frequency at about 4 Hz. The peak amplitude is getting larger and the damping is also stronger from case (a) and (b) to case (e) and (f). Therefore, the three plots generated by the function of this system are in very close agreement with the plots

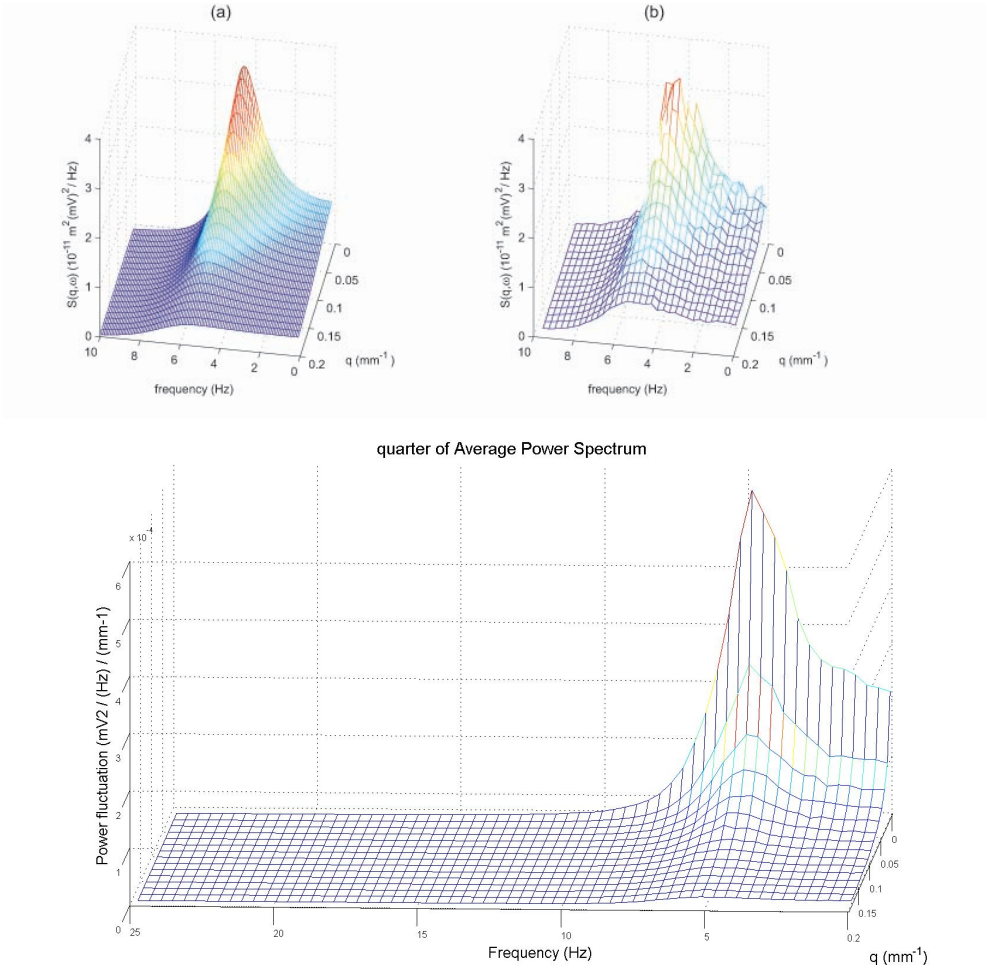


Figure 6.21: *Power Spectrum generated by this system for comparison with case (a) & (b).*
The plot (a) is the predicted power spectrum from linear analysis and the plot (b) is the simulated power spectrum produced by Wilson et al. The bottom one is the simulated power spectrum produced by this system's function for comparison with case (a) & (b).

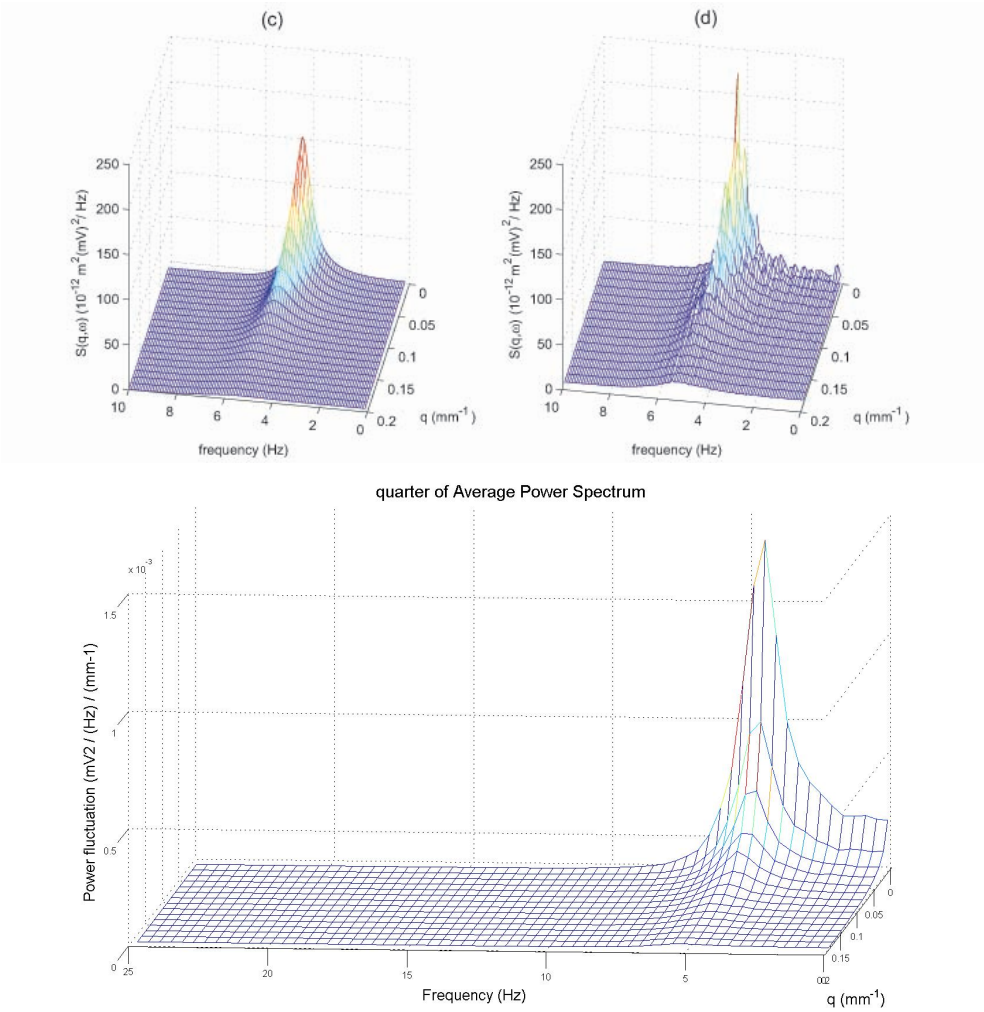


Figure 6.22: *Power Spectrum generated by this system for comparison with case (c) & (d).*
The plot (c) is the predicted power spectrum from linear analysis and the plot (d) is the simulated power spectrum produced by Wilson et al. The bottom one is the simulated power spectrum produced by this system's function for comparison with case (c) & (d).

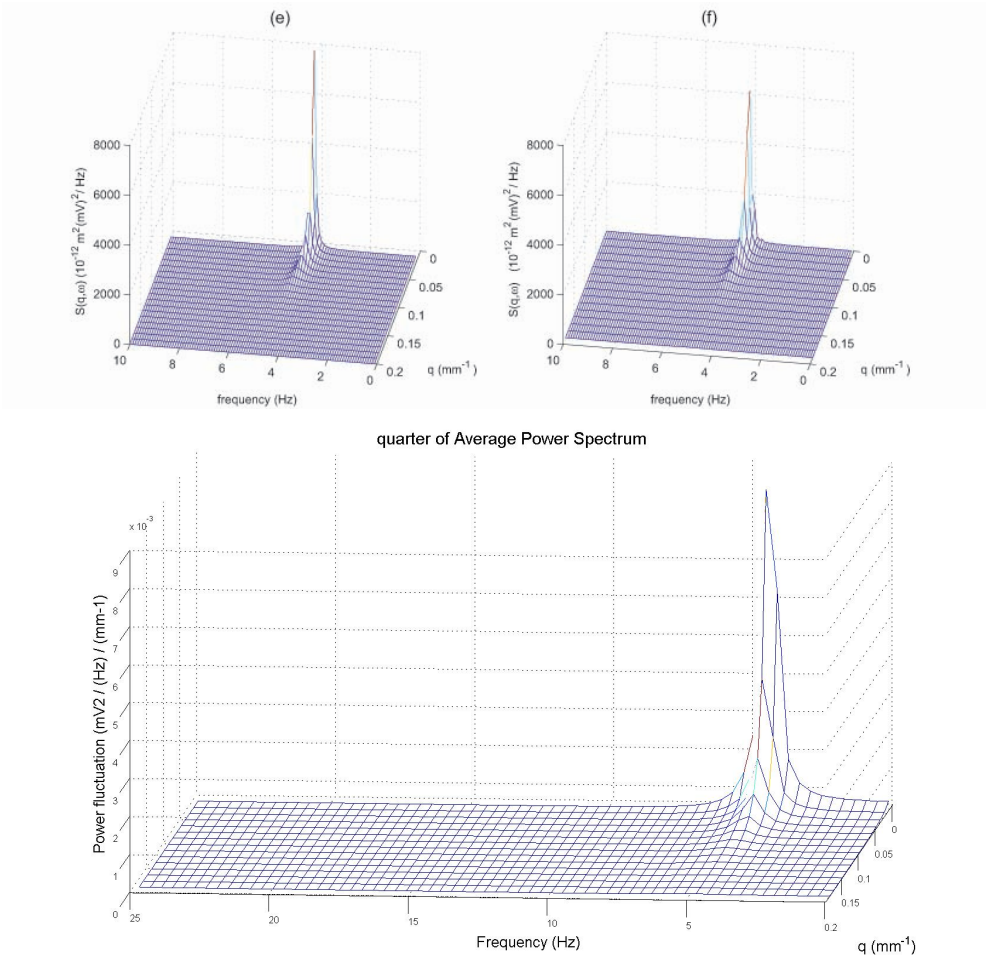


Figure 6.23: *Power Spectrum generated by this system for comparison with case (e) & (f).*
The plot (e) is the predicted power spectrum from linear analysis and the plot (f) is the simulated power spectrum produced by Wilson et al. The bottom one is the simulated power spectrum produced by this system's function for comparison with case (e) & (f).

of the paper. However, they are not the same because of the different interpretations of the power spectrum, as explained above.

6.7 Perturbation: I.kicker = 1

This function is designed to see the response of the neural signal if there is a step-change in it. This can be done by setting the perturbation control-flag to be true (`I.kicker = 1;`). Figure 6.24 shows the response of excitatory soma potential (V_e) after the perturbation is activated for a while. For the case in Figure 6.24, the step-change is applied on one point in the middle of the cortex plane initially, and the plot shows the soma potential is changing rapidly on the whole cortex plane. This means a step-change in one macrocolumn will cause a chain reaction in other macrocolumns nearby, then the V_e signal of the whole cortex will oscillate for a while before settling down (22).

6.8 Performance of the system:

Since this system involves a lot complex calculations, so it might take some time to run the simulation. This system has restrictions on the performance, due to the limits on the hardware and software of the computer. There are several things which might affect the simulation performance.

6.8.1 I.Nspace

This parameter is the number of discrete points on the coordinate axis of the cortex plane. It determines the density of the grid on the plane. The larger the value for this parameter, the higher the resolution of the simulation for the cortex plane. However, this parameter can not be set to a very large value, because it slows the simulation speed and affects the saved data. The larger `Nspace` means there are more discrete points being simulated, so it involves much more calculations. And the MATLAB has a limit for the saved file size, so MATLAB might not be able to save the data to a file if it is too large. It means if `Nspace` is larger, then less time sets of data can be saved for the simulation. Therefore, the user should input a reasonable value for `Nspace` (`Nspace = 32` is a recommended value for the simulation).

6.8.2 I.saved_variable

This parameter determines which variable to be saved. There are 16 variables processed in the simulation by the system for modified Liley model. However, the user might only be interested in one or a few of them (usually V_e). The system passes the variable data to global for saving. This passing takes time and MATLAB has a data storage limit for global. Therefore, if there are fewer variables passed to

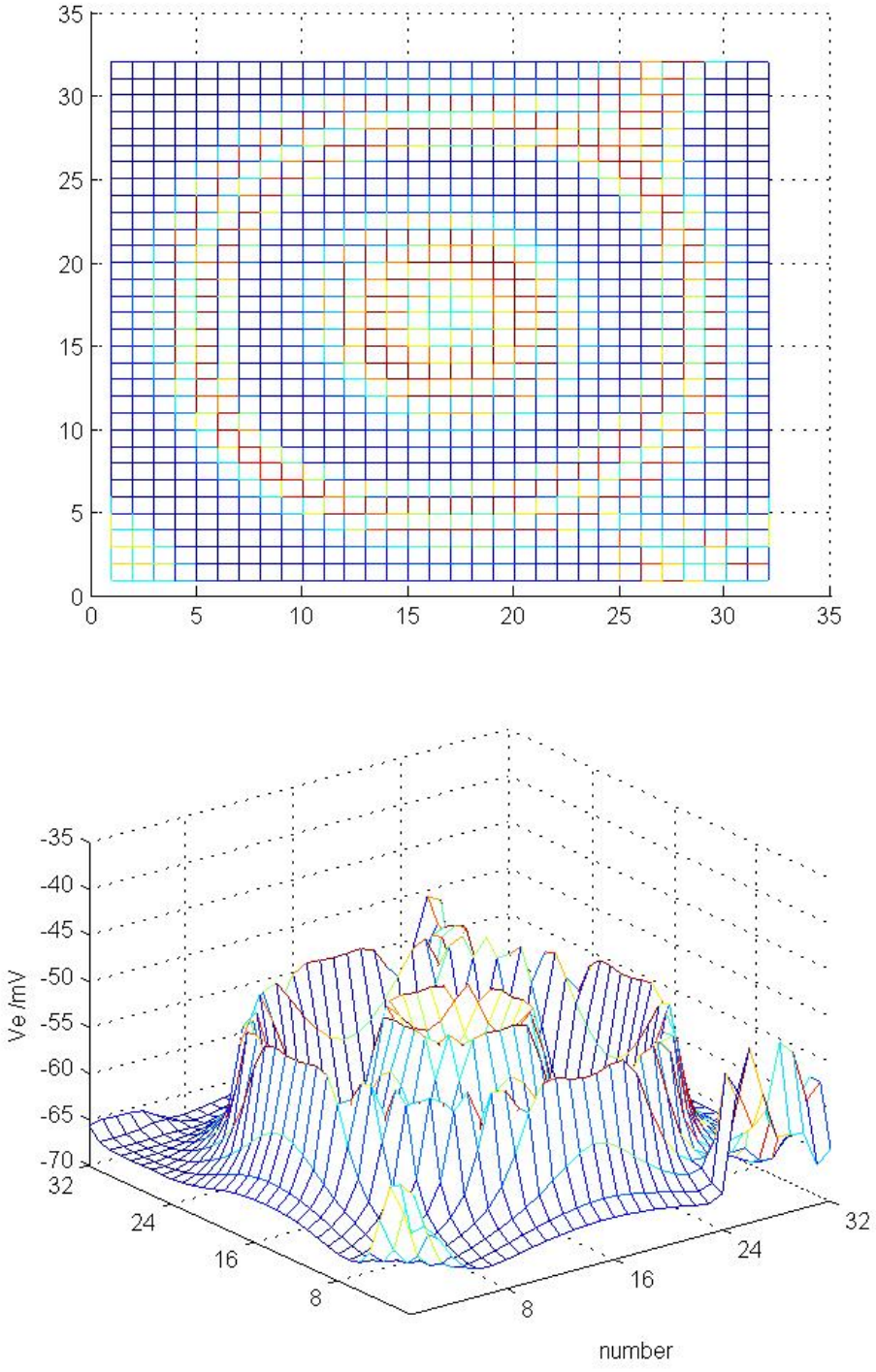


Figure 6.24: *Example of Perturbation.*
This figure shows the whole cortex is oscillating when the perturbation is activated. The perturbation in this case is activated by applying a step-change in the middle of the cortex plane.

global, then it speeds up the simulation and more time sets can be saved for the data (the length of simulation can be longer). For this reason, the user should only save variables that he requires.

6.8.3 `Tend` & `deltat`

The simulation length is determined by `tend` and the corresponding real time in seconds for the simulation length is determined by the product of `tend` and `deltat`. The simulation takes longer time for larger `tend` or smaller Δt. Therefore, if the user would like to simulate the cortex for a long period, then a larger, but reasonable value for Δt is necessary, otherwise it might take a long time for simulation. The value of Δt cannot be too large, otherwise the numerical integration methods RK2 and RK4 will become unstable. There is a minimum value Δt can have. Therefore, a reasonable value for Δt is required for the simulation ($\Delta t = 0.0002$s is generally good for a cortex of **length** 0.5 m).

6.8.4 `t_step_4_tplot` & `t_step_4_graph`

The parameter `t_step_4_tplot` determines the frequency of passing the data to global during the simulation and `t_step_4_graph` determines the frequency of plot generating. Since the actions of data passing and plot generating always take time and they slow the simulation speed significantly. However, the plot generating can be deactivated by setting `t_step_4_graph` to zero. This is useful for speeding up the simulation if the user only wants to record data without watching the changing on the cortex plane. `t_step_4_tplot` determines the time interval between the sets of data. It affects the resolution of simulation in time axis. Therefore, a reasonable value is required for this parameter (`t_step_4_tplot = 10`$\sim$`100` is generally good).

6.8.5 Numerical Integration

This system contains two methods for numerical integration: second-order and fourth-order Runge-Kutta. The user can select either one for numerical integrating the variables. However, the simulation takes longer if the RK4 (fourth-order Runge-Kutta) method is used, because RK4 involves more calculations than RK2. Therefore, a user should estimate his requirement before selecting the numerical integration method.

6.8.6 Stability

If both control-flags `dataFlag` and `stabilityFlag` are set to `true` (or 1), then the system will perform a stability test at every `t_step_4_tplot`. This is useful for analysis if there is a trajectory. However, it slows the simulation speed dramatically, because the stability test involves a lot calculation. Therefore, the user is advised

Table 6.1: Elapsed Time for each function

Function	**Elapsed Time (ms)**
Steady_States_stability_14×14	15.352
steady_states_delV_lambda	164.745
noise - Gaussian	8.547
noise - Poisson	61.322
Numerical Integration - RK2	105.130
Numerical Integration - RK4	121.405
Trajectory - Isoflurane	5.078
Pertubation	1.987
data_output (with saving and plot generating)	180.371

to deactivate the control flag of stability test if the stability of steady-states is not required by the user.

6.8.7 movieFlag

This is the control flag for generating the movie. The system records the frame of the plot and passes it to global for saving the motion pictures. However, it slows the simulation speed, because it involves data passing. Therefore, it should be deactivated if not required.

6.8.8 Elapsed Time

Table 6.1 shows the elapsed time by each function, using a typical set of input parameters.

The `Nspace` value used for getting the figures in Table 6.1 is **32** and the cpu used is AMD ML-37 2GHz. It is easy to figure out the computational resources used by each function by looking at the table above. Some functions in the system do take a lot time during the simulation compare with other functions. For example, we can see that outputting data and evaluating the steady-states take the longest time. This table provides ideas to a developer about the functions which can be considered to be reconstructed.

Chapter 7

Evaluation

The completed project is going to be examined by referring to the user-requirement.

1. *Must run on PC and MAC*
   This requirement is done. The programming language used to design this system is MATLAB. MATLAB program can be installed on both Windows and MAC, so this system is compatible for both operating systems and the user can use this system once they have installed MATLAB on their computer.

2. *Must be easy for a user to modify.*
   This system has several input functions, like trajectory, perturbation, 1^{st}-order derivative, steady-states, stability, etc... they are all kept in their own independent functions. Each function is coded in a logical way and the user can locate the part easily by defining the input function that the user is going to modify. The detail discussion for these is in Chapter 4.

3. *Full documentation, including a user-guide, must be written.*
   This is done and it has a step-by-step tutorial in it for each function. The user-guide is in Appendix A.

4. *User must be able to supply any model that can be written as a set of first-order differential equations in the form* $dy = f(y)dt + \xi$
   All the equations for the model are written in the form of first-order differential equations and these equations are used to numerical integrate the variables. They are stored in `function first_order_derivative` and it is an input function. The user can supply any model by replacing the equations here with another set of first-order differential equations.

5. *Up to 50 first order equations possible*
   There is no limit for the number of input equations, so up to 50 first-order equations are acceptable by the system. However, more input equations means there are more variables, and they will slow down the simulation speed, so the user should keep this in mind.

6. *Model should work in at least two spatial dimensions*
   The cortex in the system is assumed as a square plane with a number of discrete points in it, so the spatial components of each variable are in an `Nspace`×`Nspace` matrix for the simulation. Therefore, the model works in two spatial dimensions.

7. *User should be able to change the equations (model)*
   This can be done by changing the equations stored in `function first_order_derivative`, `function Steady_States_stability_14`×`14` and `function steady_states_delV_lambda`, and also the parameters stored in `function Input`. The details for changing the model are in the User-Guide.

8. *User should be able to change values of any parameters*
   All the parameters and control flags are stored in the function Input and the user can change the value easily.

9. *User should be able to specify how different parameters change with time*
   `Function find_trajec` is designed to manage this. The user can use the control flag in the input function to select the trajectory for the simulation. The user can also modify the present trajectory or add a new trajectory in this function.

10. *User should be able to specify a perturbation in the system at some point(s) in space and time*
    `Function perturbation` is designed for this. The user can use the control-flag in the input function to enable or disable the perturbation function. The user can also specify the position and time for perturbation in `function Input`.

11. *Software should be able to output, at the user's request:*

    - *A data-file containing $y_j(r,t)$ where j is any user-specified component of y, r is space and t is time*
      All the output parameters and variables from the system will be saved in a file with the extention **.mat** (e.g. firing_rates_etc.mat), and the file name is the name that the user specified in `function Input`. All the variables are saved in a parameter called **svec** in this file. The details of recalling the variable from the saved file are in the User-Guide.
    - *Movie file of $y_j(r,t)$ in a standard format*
      The user can enable the control-flag for saving the movie of the variable, and the file format of the movie file is AVI. AVI is a standard format for a video file and it can be opened by the most of the video player programs.
    - *Plots of $y_j(r_a,t)$ where j is any user-specified component of y and r_a is some user-specified point in space*

This is done by `function Variable_Variation`. This function loads the saved file from the main function and plots the variable vs. time. The user can see the fluctuation of a variable from this plot.

- *A power spectrum (power against temporal frequency) for the system*

- *A spectrogram plot (power against frequency and time)*

- *Plots of power against temporal frequency and spatial frequency*
  These are done by `function Power_Spectrum`. This function loads the saved file from the main function and generates these plots.

- *Stationary (equilibrium) state(s) of the system, and their stability, as a function of time*
  `Function find_steady_states` is designed to provide these data. It can be activated by setting its control-flag in the Input function. The detail for doing this are in the User-Guide.

12. *No third-party software required for which the University of Waikato does not already possess (and intends to keep renewing) a license.*
    This software is coded by using MATLAB and the University of Waikato has a license for it.

13. *User should not need to change the main code in order to run the software*
    The functions `Cortex_System` and `Runge_Kutta_Integration` are designed to be fixed and the user will not need to change or modify these functions while using this software.

14. *Should be easy to modify the code to use different numerical integration methods*
    There are two numerical integration methods in `function Runge_Kutta_Integration`. They are: second-order Runge-Kutta and fourth-order Runge-Kutta. The user can easily switch the method by setting its control-flag in `function Input`.

15. *No restrictions on the use of the software by the University of Waikato for academic research purposes.*
    The copyright of this software belongs to the University of Waikato, so there is no restriction for the staffs and students in the University of Waikato to use this software.

16. *Ensure, where feasible, that all numerical routines can be tested against standard* ***calibration*** *suite.*
    There are two functions which have a built-in test mode, they are `function Steady_States_stability_14x14` and `function Power_Spectrum`. These two

functions are easily to have bugs in their code, so they have built-in standard cases which can be used to compare the results. Therefore, they are designed to have a testing mode in it.

All the user requirements have been met. The user can run the simulation easily by using this system and get the data they require. However, the research in this area is still in progress. There will no doubt be improvements that will be made in the future. Some of these are discussed in next chapter.

Chapter 8

Future Work and Improvement

This software so far still needs to be ran under the MATLAB window. This makes the software not that easy to use, because the user needs to have MATLAB installed in the PC before using this software. Therefore, a private window with a nice and friendly user-interface can be designed for this software in the future if possible, so this software can be run in its own window without using any other program.

MATLAB has a data storage limit, and it limits the quantity of data saving by the simulation. This means this system can not perform a high resolution simulation for a long simulation period. The simulation is also very slow. Table 8.1 shows time taken for different simulation cases.

The time step-size is 0.0002s and total time steps is 5000 for the cases in Table 8.1, which means the system is simulating the cortex with one second duration, and the table shows that this **one second duration** takes about 3∼15 minutes real time (by the cpu AMD ML-37, 2GHz). This means the user sometimes need to wait for several hours to get the simulation data. It is because this system involves many calculations for the simulation and it slows down the performance of a computer. These problems can probably be solved by a faster computer in the future or a new and better numerical integration method if it can be discovered or if the equations of the model can be further simplified.

This system has the same equations three different times in its code; they are recalled once for the time-step integration (`function first_order_derivative`), once for finding the stationary states (`function find_initial_states`) and once for working out the eigenvalues (`function Steady_States_stability_14`×`14`). The equations in these functions relate to the same model, but the form of the equations in each function is different to others due to the different purposes of each. This makes it quite difficult to combine the equations in these functions into a general form. For example, it is hard to write a generic solver that will solve all differential

Table 8.1: Elapsed Time for different cases
This table shows the settings of the parameters for each case and the time taken by the simulation.

Simulation Case	**Default**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Nspace	32	64	32	32	32	32	32	32	32	32
Time Step-Size Δt (seconds)	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002
Time Steps	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000
Numerical Integration Method	RK2	RK2	RK4	RK2	RK2	RK2	RK2	RK2	RK2	RK2
Data Saving	all	all	all	all	all	all	V_e	all	all	Disable
Stability Test	enable	enable	enable	enable	enable	disable	enable	enable	enable	enable
t_step_4_tplot	10	10	10	100	10	10	10	10	10	10
t_step_4_graph	100	100	100	100	100	100	100	100	Disable	100
Noise	Poisson	Poisson	Poisson	Poisson	Gaussian	Poisson	Poisson	Poisson	Poisson	Poisson
Trajectory	Isoflurane	Isoflurane	Isoflurane	Isoflurane	Isoflurane	Isoflurane	Isoflurane	Disable	Isoflurane	Isoflurane
Elapsed Time (seconds)	*322.39672*	*831.009364*	*353.19013*	*195.974658*	*175.542563*	*279.174979*	*246.229863*	*318.6943*	*295.189696*	*194.480965*

equations to give steady-states. However, it will be a good approach to reduce this duplication by writing these equations into a general form, so the user won't need to modify the equations three times for changing the model.

One objective for the design is that the user can change the model easily for the system, so the procedure of model changing has been simplified to be as easy as possible. However, model changing of the system is still the most difficult part for using this software. Therefore, a model that fits all the cases is desirable. This would be a major goal for the future, but would be hard to achieve.

Chapter 9

Conclusion

When many different computer codes have to be written for different applications, it can be inefficient to the user. This system is developed to solve this problem for mean-field models of the cortex. This system combines many functions and can be used for many applications. Therefore, this system is a versatile piece of software with an easy-to-use code and increases the efficiency for the cortical simulation.

Since all the user requirements have been met, this project has gone quite well overall. However, there were some difficulties in developing this system. The major difficulty was the mathematics. The major equations of the model are either first or second order differential equations. There are a lot calculations involved and they have to be done on paper before coding them. For example, the calculations for finding the steady-states and their stability are quite complicated. Silly mistakes sometimes happened and they cause errors in the calculations, which make the results inaccurate. However, with help from M.T.Wilson and D.A.Steyn-Ross, these problems are solved and the system is now working correctly. And this project made me learn a lot about the techniques in project organizing, software design processing, mathematics and biophysics for cortex modelling.

Therefore, this software now can simulate the response of the neurons in the cortex for many circumstances. The user can examine the result of the simulation and analyze the data. A trajectory (e.g. for applying an anaesthetic drug) can be added to the system easily. It may be useful for understanding how some drugs work. It is anticipated that the software will be useful for the sleep and anaesthetic research.

Appendix A: User-Guide

A.1 Introduction

The cerebral cortex consists of neurons. A Neuron is a nerve cell, which receives and transmits electrical signals. These signals are significant to a human's behaviour. Since the neurons cause currents, and these currents produce electrical fields, so these neural signals can be measured by the scalp electrodes of an electroencephalography (EEG). These EEG signals will be changed for different behaviours. As long as the brain is not dead, then the spontaneous activities of neurons will produce a series of EEG signals. Therefore, many neuron models have been developed for simulating the cortical signal, and each model is developed for a different purpose or application. A different computer code has to be written for different model, and it can be inefficient, so this system is developed to allow the simulation of cortical signals by any model with a mean-field approach. This system brought together various research codes written previously into a single, versatile and easy-to-use piece of software. It can be used for any general mean-field model.

This tool has the following functions:

- Simulates the cortex through an user-defined system of equations (user-defined mean-field model)
- Displays the dynamic signal
- Saves the data
- Processes the output through spectrograms, etc...
- Evaluates equilibrium solutions and their stability
- Can perturb the system in a controlled way

The whole system can be divided into four sections: Input, Main, Output and Output Processing.

Input This system contains several input files. It is for the user to change the input or model easily.

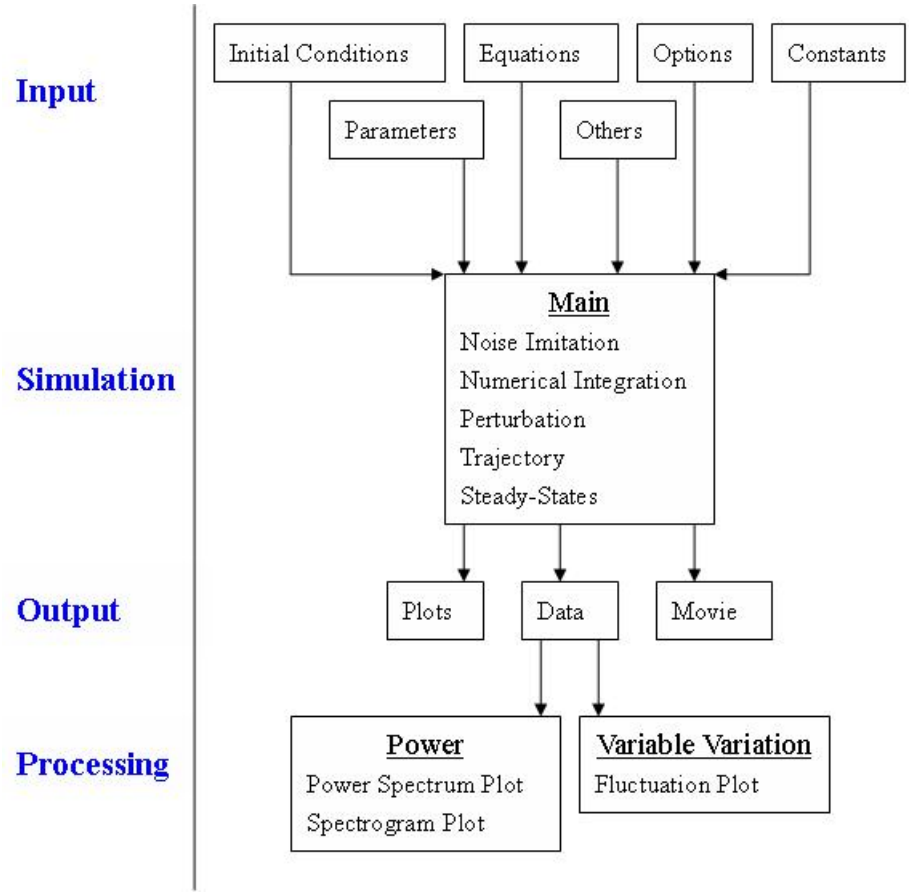


Figure A.1: *Structure of the system*

Main The software simulates the cortex by numerically integrating the variables in the equations of the model. The numerical integration method used for integrating the variables is based on Runge-Kutta algorithm. It is to simulate the code with any mean-field model that can be written as $\frac{dx}{dt} = f(x,t) + \xi$, where x is a multidimensional vector, f is an user-defined function and ξ is a noise term. The cortex is simulated with a 2 dimensional grid, so the user can easily monitor the variables during simulation.

Output The system outputs the data, plots and movie to the user.

Output Processing The output data can be further processed by the output processing functions. The plots will be generated for the simulation data and the user can analyze the cortical activities based on these plots.

Figure A.1 shows the structure of this software system.

A.2 System Requirement

This tool is developed by using MATLAB. MATLAB must be installed before starting using this tool.

A.3 Input Procedure for Simulation

A.3.1 function Input

This is the user input file. This file contains model parameters, control flags, graph labels & file names, etc. The user can change the parameter value, control flag, etc... for simulating the cortex system. These inputs can be sorted into two groups: generic and specific.

Generic Input

These are associated with the generic operation of the simulation and they are independent to the model of the simulation. The changes for these generic inputs will change the operation of the simulation, such as time-length or time step-size of the simulation.

Input Parameters & Control Flags, etc.	Description
I.lcortex	The length of the cortex. (The cortex here is assumed as a square plane)
I.Nspace	Number of discretisation points, this defines the modelling resolution of cortex.
I.deltax	The increment size of cortex length for modelling. It is defined by dividing cortex length by number of discretisation points.
I.deltat	Time step-size for simulation, unit in second.
I.tend	Total time steps for simulation.
I.RK	Control flag for choosing 2nd order or 4th order Runge-Kutta. 2 = RK2 and 4 = RK4.
I.FUN	Function name of differential equations for numerical integration.
I.branch	Control-flag for selecting starting point from the branch for the steady-state. 0 = bottom branch; 1 = upper branch.
I.trajectory	Flag for trajectory, where the trajectory is the path of time changing for the parameters. 0 = no path; 1 = drug trajectory; 2 = user path.
I.poissflag	Control-flag for noise imitation. 0 = gaussian; 1 = poisson.
areapermac	Area of macrocolumn.
I.macsperbrain	Number of macrocolumns for the simulation.
I.kicker	Control-flag for perturbation, where it step-changes the cortical signal in a user-defined way. true = activate; false = deactivate.
I.kick_division_period	Division period for perturbation.
I.kickTime	Time length for perturbation.
I.kickVoltage	Perturbation voltage.
I.x_area	Perturbation area
I.y_area	
I.t_step_4_tplot	Number of time steps for the interval of per data recording.
I.t_step_4_graph	Number of time steps for the interval of plotting graph.
I.plotted_variable	The user-selected variable for plotting.
I.saved_variable	The selected variables for saving.

I.x_axis_range	Set limitation for axes.
I.y_axis_range	
I.z_axis_range	
I.xtick	Set tick position on axes.
I.ytick	
I.x_axis_label	Axes label.
I.y_axis_label	
I.z_axis_label	
I.dataFlag	Control-flag for saving the data. true = enable; false = disable.
I.movieFlag	Control-flag for saving the movie. true = enable (t_step_4_graph must NOT be 0); false = disable.
I.stabilityFlag	Control-flag for performing stability test. true = enable stability test. false = disable stability test.
I.DataFileName	File name for data output
I.MovieFileName	File name for movie output

Specific Input

These inputs are associated with the model of the simulation. They must be modified if the model is changed. The system currently uses a modified Liley model as the example set of equations. The parameters for this model are:

Input Parameters & Control Flags, etc.	**Description**
I.lambda_Ach	λ_{Ach}: Effect of muscarinic neuromodulation
I.lambda_i	λ_i: Multiplier on inhibitory synaptic weight
I.gamma_i	$\gamma_{ie,ii}$: Inhibitory synaptic rate constant
I.gamma_e	$\gamma_{ee,ei}$: Excitatory synaptic rate constant
I.lamda_ee	$\Lambda_{ee,ei}$: Characteristic inverse length-scale for connections
I.del_Vrest	ΔV_e^{rest}: Effect of altering extrasynaptic ion channels
I.nu	Relative amount of noise. 1 = pure poisson sc terms; 0 = pure eqm subcortical terms
I.v	υ: mean axonal conduction speed
I.q	∇
I.graphFlag	Control flag for plotting graph of steady-state error residual. 1 = enable graph plotting; 0 = disable

A.3.2 Procedure

There are some parameters and control flags have to be checked before starting simulation.

1. **Inputs the temporal parameters**

   `deltat`: Δt, *RK time step-size (unit in seconds)*
   e.g. `I.deltat = 0.2*1e-3;`

   `tend`: *number of time-steps*
   e.g. `I.tend = 5000;`

   The variables in the model will be numerically integrated step by step, so `tend` is the number of total time steps for the integration rather than the value of time-length. Therefore, the corresponding actual time for the simulation can be calculated by multiplying `deltat` with `tend` (unit in second). e.g. If `deltat = 0.0002` and `tend = 1000`, then the system will simulate the fluctuation of the cortical signal for 0.2 seconds period with a time step-size of 0.2 ms.

   Warning:

   - Many time steps will lead to a long run-time.
   - MATLAB has a data storage limit. The data can not be saved if the amount of data exceeds the limit.

   Data should be saved less frequently to avoid the restrictions above. Therefore, the user also needs to specify the parameters:

   `t_step_4_tplot`: *user-defined No. of t steps for the interval between each saved data set*
   e.g. `I.t_step_4_tplot = 10;`

   Due to concern the performance of the simulation, the data might not be saved for every step of the simulation. This parameter defines how frequently for saving the simulation data, where **tplot** is the count for the time sets increment of the data-saving. The larger `t_step_4_tplot` value means the data will be saved less frequently, but it increases the simulation speed. The actual time in seconds for `t_step_4_tplot` can be calculated by `deltat`×`t_step_4_tplot`. e.g. If `deltat = 0.0002` and `t_step_4_tplot = 100`, then the actual time for the interval between each set of the data is 20ms.

`t_step_4_graph`: *user-defined No. of t steps for the interval of plot generating (No graph will be generated if set to 0, for faster simulation)*
e.g. `I.t_step_4_graph = 100;`

This parameter defines how frequently for generating the plots. This is for the user to monitor the variance of the cortical signal during the simulation and also for the frames of the movie. The larger `t_step_4_graph` value means the plot will be generated less frequently, but it increases the simulation speed. If `t_step_4_graph` is set to 0, then no graph will be plotted, but the simulation can still run. (For faster simulation speed)

2. **Inputs the function name of the file stores first order derivatives equations for the variables and selects the numerical integration method.**

`FUN`: *Name of the function which stores the* 1^{st}*-order derivative equations of the model*
e.g. `I.FUN = 'first_order_derivative';`

`RK`: *Control-flag for selceting the numerical integration method based on Runge-Kutta algorithm*
e.g.
2nd-order Runge-Kutta: `I.RK = 2;`
4th-order Runge-Kutta: `I.RK = 4;`

The numerical integration method this system uses is based on Runge-Kutta algorithm (18); either second (`I.RK = 2;`) or fourth-order (`I.RK = 4;`). Runge-Kutta method can be selected for the integration.
RK4 method provides a more accurate approximation than RK2, but RK4 simulation is slower than RK2.

3. **Inputs the number of discretisation points for the simulated cortex plane.**

`Nspace`: *number of x and y coordinates*
e.g. simulation on a 16×16 grid.
`I.Nspace = 16;`

This defines the resolution of the simulation for the cortex plane. The larger `Nspace` value means the higher resolution of the simulated cortex plane, but it also slows down the simulation speed. The users' choice will also be limited by the data-storage-limit for the saved data. The user cannot input a value which

is too large. Therefore, a reasonable value is required for this one (`Nspace = 32` is recommended).

4. **Select a trajectory for simulation.**

   `trajectory`: *control-flag for selecting the trajectory*
   e.g.
   no path: `I.trajectory = 0;`
   drug path: `I.trajectory = 1;`

   Some parameters might be time-dependent in some circumstances. For example, a drug has been taken by a patient. The trajectory function is a user-defined function that controls how certain variables change with time, and the trajectory control-flag activates this function.

5. **Select either Poisson or Gaussian for noise imitation.**

   `poissflag`: *control-flag for selecting the probability distribution used to generate the random arrays for noise imitation*
   e.g.
   `I.poissflag = 0;`
   Gaussian noise imitation; this means the noise will be imitated by the random arrays generated from a normal distribution.
   `I.poissflag = 1;`
   Poisson noise imitation; this means the noise will be imitated by the random arrays generated from the Poisson distribution. Poisson is more physiologically plausible, but Gaussian is much faster.

6. **Select either enabling or disabling perturbation while simulating.**

   `kicker`: *control-flag for step-changing the cortical signal manually*
   e.g.
   Enable: `I.kicker = 1;`
   Disable: `I.kicker = 0;`

   If the flag is enabled, then the following parameters need to be concerned, otherwise they can be ignored.

   `kick_division_period`
   This parameter determines how frequently the cortical signal has been step-changed during the simulation (Unit in seconds).

`kickTime`
This parameter determines the time of duration for each step-change during the simulation (Unit in seconds).

`kickVoltage`
This parameter determines the amount to step-change the cortical signal (Unit in mV).

E.g. to step-change the parameter ΔV_e^{rest} by 10 mV with 2 seconds duration for every 5 seconds:

```
I.kick_division_period = 5;
I.kickTime = 2;
I.kickVoltage = 10;
```

`x_area`
`y_area`
These two parameters determine the position for beginning kick-change. This position must be within the `Nspace`×`Nspace` grid.
User can either apply the perturbation on only one point or on an area on the cortex plane grid.

To apply perturbation on only one point, the user inputs the desirable values for `x_area` and `y_area`.
e.g.

```
I.x_area = 12;
I.y_area = 16;
```

To apply perturbation on an area, the user inputs the ranges for x and y.
e.g.

```
I.x_area = 12:16;
I.y_area = 16:18;
```

7. **check the values of the following parameters.**

The parameters below are for modified Liley model. They can be modified if the model is changed.

`lambda_Ach`
`lambda_i`
`gamma_i`
`gamma_e`

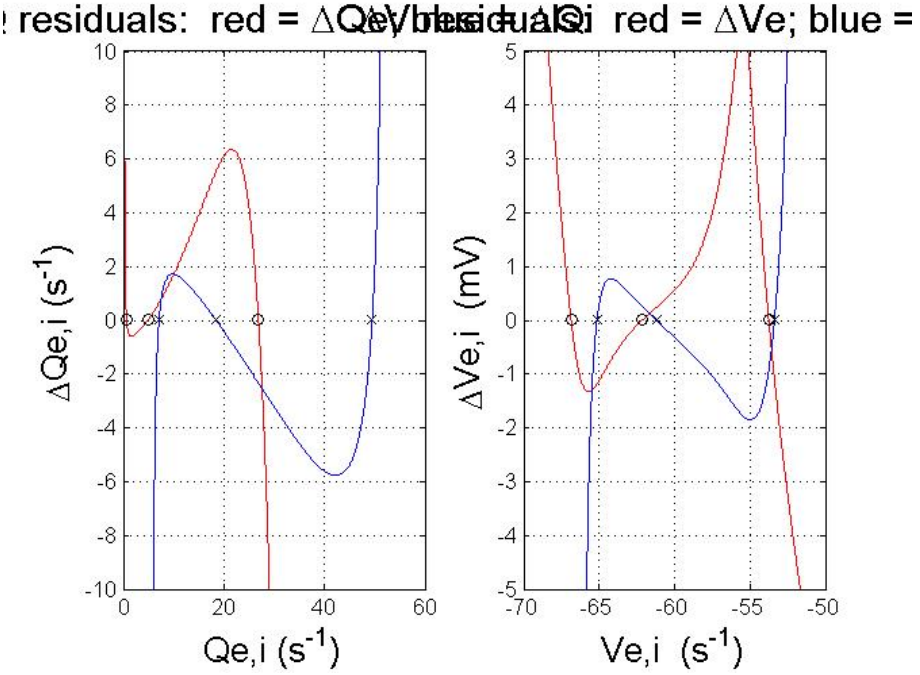
`lamda_ee`
`del_Vrest`
`nu`
`v`
`q`

8. **Select either bottom or top branch of stationary states to start with.**

   `Branch`: control-flag for selecting the branch of the stationary states
   Bottom branch: `I.branch = 0;`
   Upper branch: `I.branch = 1;`

   This parameter defines the branch of the stationary states for the initial condition of the variables. If there is only one steady-state, then the setting of `Branch` will not have any effect, because both upper and bottom branches return the same value.

   If the `graphFlag` (`I.graphFlag = 1;`) is enabled, the solver for the stationary states for the modified Liley model will produce a graph showing the accuracy of the solution (the so-called 'steady-state error residuals)



9. **Select variables (1-14) for plotting and variables for recording.**

   There are 14 variables for the modified Liley model, and they are defined as:

Index	Varaible	Description
1	V_e	Excitatory soma potential
2	V_i	Inhibitory soma potential
3	Φ_{ee}	Postsynaptic Flux from Exhibitory to Exhibitory Neurons
4	X_{ee}	Rate of Change of Φ_{ee}
5	Φ_{ei}	Postsynaptic Flux from Exhibitory to Inhibitory Neurons
6	X_{ei}	Rate of Change of Φ_{ei}
7	Φ_{ie}	Postsynaptic Flux from Inhibitory to Exhibitory Neurons
8	X_{ie}	Rate of Change of Φ_{ie}
9	Φ_{ii}	Postsynaptic Flux from Inhibitory to Inhibitory Neurons
10	X_{ii}	Rate of Change of Φ_{ii}
11	ϕ_{ee}	Presynaptic Flux from Exhibitory to Exhibitory Neurons
12	Y_{ee}	Rate of Change of ϕ_{ee}
13	ϕ_{ei}	Presynaptic Flux from Exhibitory to Inhibitory Neurons
14	Y_{ei}	Rate of Change of ϕ_{ei}

`plotted_variable`: *parameter for selecting a variable to be plotted.*
The user can monitor the variance of a variable by setting this parameter.
e.g. to select the **excitatory soma potential** for plotting.

```
I.plotted_variable = 1;
```

`saved_variable`: *parameter for selecting the variables to be saved.*
Each variable is an `Nspace`×`Nspace` matrix and it has a number of `tplot` time sets for its data, where `tplot` is a number of time step increment for saving. The data of the variables will be saved in a vector called `svec`. It is a 4-D (`Nspace`×`Nspace`×`saved_variable`×`tplot`) matrix and its structure is svec(x-coordinate, y-coordinate, variable, time). For example, if it is a 32×32 grid simulation with 1000 time sets and there are two variables being saved, then the dimension for its `svec` matrix will be 32×32×2×1000. This will be further discussed in Section A.4.
To save the variables, user can either save all 14 variables, only one variable, or several selected variables.

To save all variables:

```
I.saved_variable = ':';
```

To save one variable (e.g. to save V_e):

```
I.saved_variable = 1;
```

To save several selected variables (e.g. to save V_i, Φ_{ei} and Φ_{ii})

```
I.saved_variable = [2 5 9];
```

(**Warning**: be concerned about the data storage limit while saving the data of variables.)

10. Inputs the limits, ticks and labels of axis for the plot.

```
x_axis_range
y_axis_range
z_axis_range
```
e.g. `I.x_axis_range = [1 I.Nspace];`

```
xtick
ytick
```
e.g. `I.xtick = [0, 0.25, 0.5 0.75, 1.0]*I.Nspace;`

```
x_axis_label
y_axis_label
z_axis_label
```
e.g. `I.z_axis_label = 'Ve /mV';`

11. Select either enabling or disabling data saving, movie generating and stability testing of stationary states.

`dataFlag`: *Control flag for enabling or disabling data saving*
`movieFlag`: *Control flag for enabling or disabling movie saving*
`stabilityFlag`: *Control flag for enabling or disabling stability test*
e.g.
`I.dataFlag = true (or 1)` : enable data saving
`I.dataFlag = false (or 0)` : disable

The table below shows how the setting of `dataFlag` and `stabilityFlag` affects the saved data:

`dataFlag`	`stabilityFlag`	**Data saving description**
False	—	No data will be saved.
True	False	Only the initial steady-state and its stability will be saved.
True	True	All the steady-states and their stabilities will be saved, at every time step. Useful when there is a trajectory. (**Warning**: This slows down the simulation speed.)

Inputs the file name for the saved data (`DataFileName`) if the `dataFlag` is enabled, otherwise it can be ignored.
Inputs the file name for the saved movie (`MovieFileName`) if the `movieFlag` is enabled and `t_step_4_graph` is not set to 0, otherwise it can be ignored.
e.g. `I.DataFileName = 'firing_rates_etc';`

If the parameters and control flags above are all set, then the simulation can be started.

A.4 Simulation

This system performs grid simulation on the full set of equations in two-dimensional space.

1. Make sure that the main function and all the input functions are in the same folder before starting the simulation.

2. Change the **Current Directory** to the work folder.

3. Type in main function name `Cortex_System` in the Command Window, then press enter. The simulation should be started now.

The system generates two plots (Figure A.2) while simulating. These two plots show the real time fluctuation of the cortical signal during the simulation. The colour of the grid shows the difference of the variable value between the discrete points on the cortex plane. The red colour represents the larger value and the blue colour represents the smaller value of the variable. The vertical axis in 3-D view shows the exact value of the plotted variable.

And the MATLAB Command Window displays the progress for the simulation(Figure A.3). The user can estimate the speed and remaining time of the simulation based on this information.

The saved data can be loaded in the Command Window, where filename is the file specified by `I.DataFileName` (e.g. load firing_rates_etc), then the saved parameters / variables will be showed in Workspace (Figure A.4).

Since the variables (V_e, V_i, Φ_{ee}, etc...) are saved in the matrix called `svec`

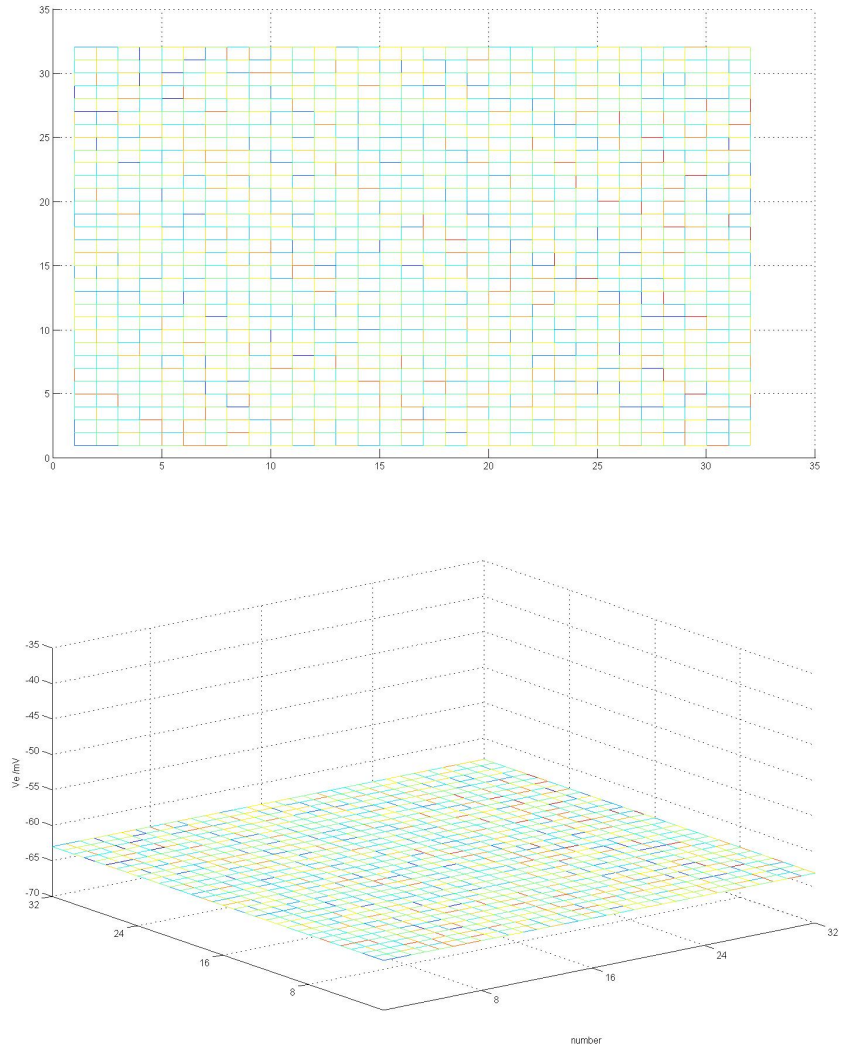


Figure A.2: *Example of simulated cortex plane plot*
The top one is the plot in 2-D view and the bottom one is in 3-D view. The red colour of the grid represents the higher cortical signal strength and the blue colour represent the lower signal strength. The difference of cortical signal strength between each each grid can be clearly observed from vertical axis of the 3-D view plot.

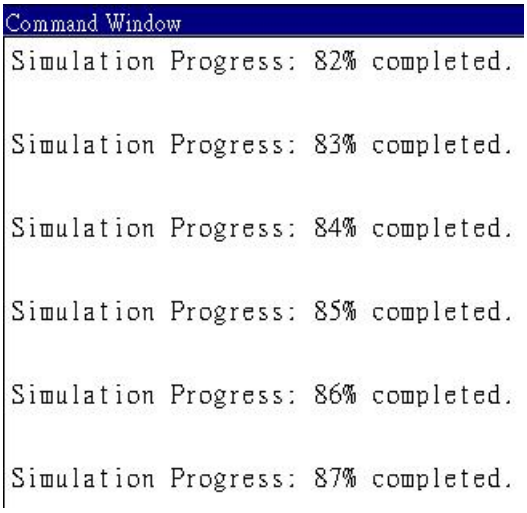


Figure A.3: *Percentage of the simulation progress in MATLAB Command Window The simulation progress can be tracked by this.*

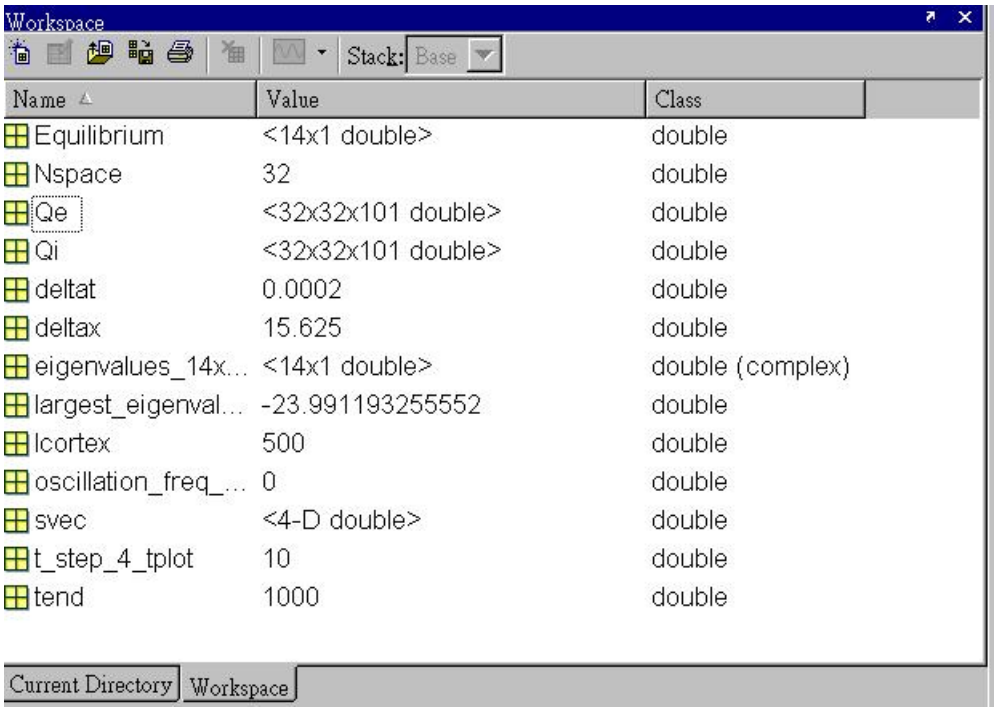


Figure A.4: *Example of Workspace with loaded data*
The data of parameters and variables can be checked in MATLAB Workspace window after calling the saved data file.

and `svec` is a 4-D (`Nspace`×`Nspace`×`saved_variable`×`tplot`) matrix, where its indices are `svec(x, y, variable, time)`. Therefore, for example, if V_e is the first `saved_variable`, then we can type V_e(:,:,:) = `svec`(:,:,1,:) in the Command Window to get V_e data in 3-D (`Nspace`×`Nspace`×`tplot`) matrix form.
x and y in the svec are the coordinates of the simulated cortex plane. Because the plane is a `Nspace`×`Nspace` grid, so the value for x and y is ranging from 1 to `Nspace` value.
If `dataFlag` is **true** and `stabilityFlag` is **false**, then only one set of stability data (initial steady-states) will be recorded.
If `dataFlag` and `stabilityFlag` are both set to **true**, then all the steady-states and stabilities data will be saved, which results in *tplot+1* sets of data for equilibrium, eigenvalue, largest eigenvalue and oscillation frequency. (ONE more time set of data than the saved variables.) The first set of data for equilibrium, eigenvalue, largest eigenvalue and oscillation frequency is the initial one (t = 0). The second set of data will be the same time axis as *tplot = 1*, and so on... The reason for this is because the recording of equilibrium, eigenvalue, largest eigenvalue and oscillation frequency starts at $t = 0$, and the recording of variables starts at $t = tplot$. This action might result a huge amount of data and it slows the simulation speed dramatically, so the user should think carefully about the necessity of this action before doing it.

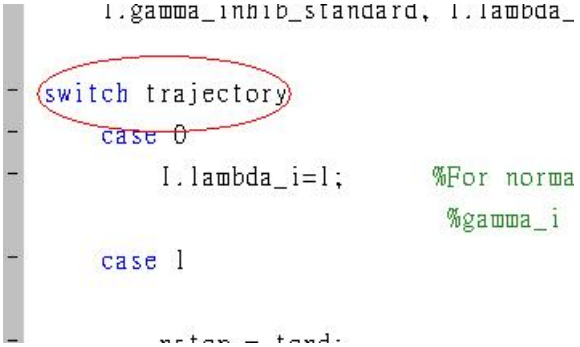
A.5 Modifying Input Functions

A.5.1 `function find_trajec` (find_trajec.m)

This function controls how the parameters changes with time.
This function so far has three trajectories:

1. no path (0)
2. drug-path (1)
3. user-path (2)

The existent trajectory can be modified or new trajectory can be added to the function. This can be done by adding new case under the switch statement in the code (find_trajec.m).



E.g. to make the value of parameter λ_{Ach} decrease smoothly by 0.25 through the course of the simulation:
`I.lambda_Ach = lambda_Ach - 0.25/tend;`

A.5.2 function perturbation (perturbation.m)

This function step-changes the variable and the way of this step-change can be controlled manually by the user. The perturbation should be able to be seen(Figure A.5) when it is activated (`kicker = true`).

The way for applying perturbation can be changed by modifying the code in the function (perturbation.m).
E.g. to step change the parameter ΔV_e^{rest} by 10 mV with 2 seconds duration for every 5 seconds

```
if (mod(t*deltat-1,5) < 2)
   I.del_Vrest_mat(x_area,y_area) = del_Vrest + 10;
else
   I.del_Vrest_mat(x_area,y_area) = del_Vrest;
end
```

A.5.3 function noise (noise.m)

The noise has been added to the signal to simulate a more real signal. The way of noise imitation can be modified in the file (noise.m). E.g. the noise can be imitated by the random numbers generated from the probability distributions. MATLAB build-in functions `poissrnd` and `normrnd` can be used.

A.6 Changing model

There are three functions that must be modified to change the model, because they stores definitions / equations of the model. And they are:

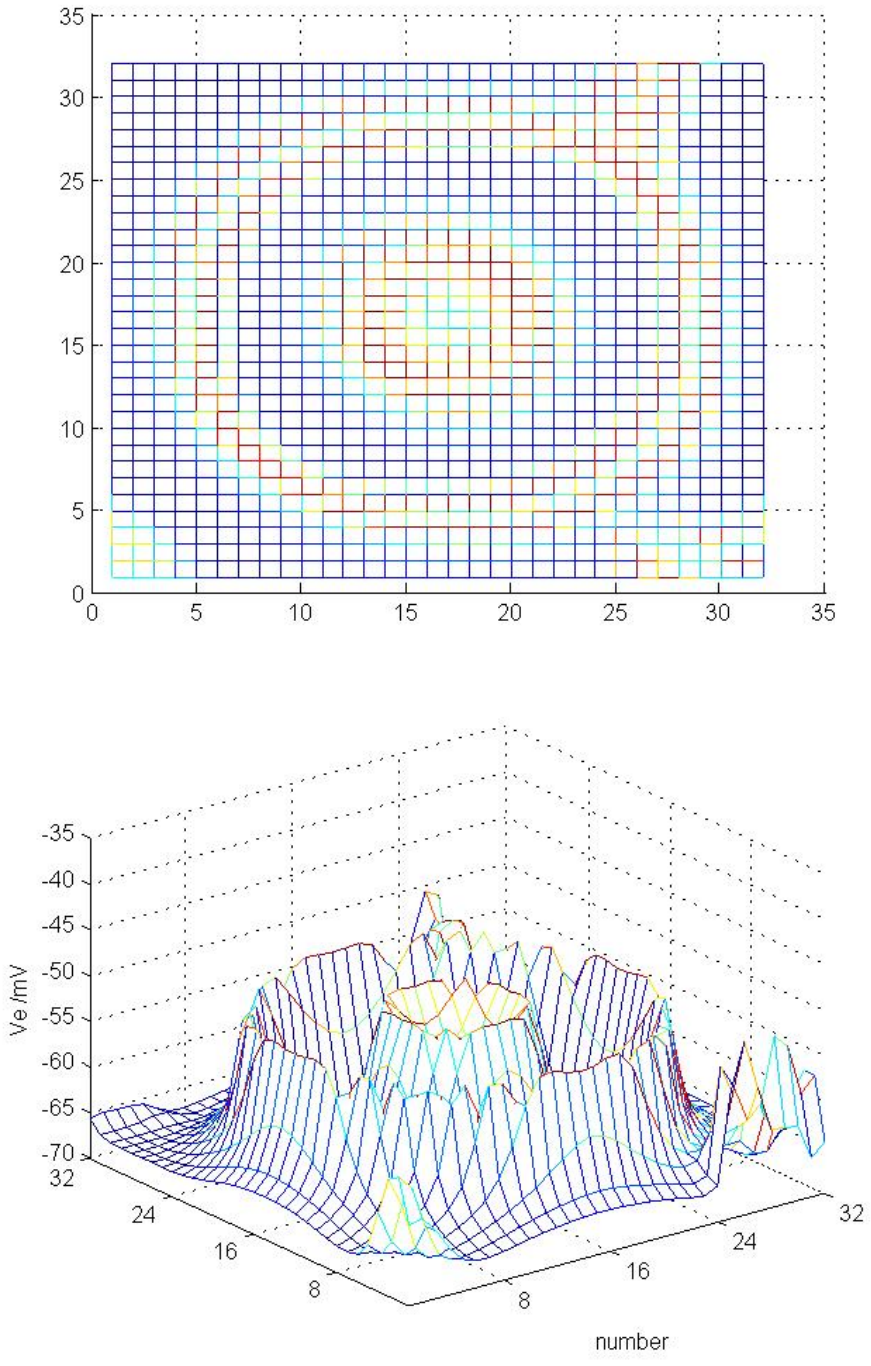


Figure A.5: *Example of Perturbation activated plot*

- `function first_order_derivative`
- `function find_steady_states`
- `function Steady_States_stability_14x14`

In addition to these, the input file containing the values of the specific variables must also be changed.

A.6.1 `function first_order_derivative` (first_order_derivative.m)

The equations of the model must be written in terms of first order differential equations, and this function stores these 1^{st}-order derivative equations. This function will be called by the numerical integration function to integrate forward the variables. The procedure of this function is:

1. Read in the constants, parameters, noise and previous state of the variables
2. Calculates the first order derivatives of the variables
3. Return the first order derivatives of the variables

To write the equations in terms of first-order equations: For example, if the equations of the model are (2): (Modified Liley model)

$$\tau_e \frac{dV_e}{dt} = V_e^{rest} - V_e + \lambda_{Ach}\rho_e\psi_{ee}\Phi_{ee} + \lambda_i\rho_i\psi_{ie}\Phi_{ie} \tag{A.1}$$

$$\tau_i \frac{dV_i}{dt} = V_i^{rest} - V_i + \lambda_{Ach}\rho_e\psi_{ei}\Phi_{ei} + \lambda_i\rho_i\psi_{ii}\Phi_{ii} \tag{A.2}$$

$$(\frac{d^2}{dt^2} + 2\gamma_{ee}\frac{d}{dt} + \gamma_{ee}^2)\Phi_{ee} = \gamma_{ee}^2(N_{ee}^{\alpha}\phi_{ee} + N_{ee}^{\beta}Q_e + \phi_{ee}^{sc}) \tag{A.3}$$

$$(\frac{d^2}{dt^2} + 2\gamma_{ei}\frac{d}{dt} + \gamma_{ei}^2)\Phi_{ei} = \gamma_{ei}^2(N_{ei}^{\alpha}\phi_{ei} + N_{ei}^{\beta}Q_e + \phi_{ei}^{sc}) \tag{A.4}$$

$$(\frac{d^2}{dt^2} + 2\gamma_{ie}\frac{d}{dt} + \gamma_{ie}^2)\Phi_{ie} = \gamma_{ie}^2(N_{ie}^{\beta}Q_i + \phi_{ie}^{sc}) \tag{A.5}$$

$$(\frac{d^2}{dt^2} + 2\gamma_{ii}\frac{d}{dt} + \gamma_{ii}^2)\Phi_{ii} = \gamma_{ii}^2(N_{ii}^{\beta}Q_i + \phi_{ii}^{sc}) \tag{A.6}$$

$$(\frac{\partial^2}{\partial t^2} + 2\upsilon\Lambda_{ee}\frac{\partial}{\partial t} + \upsilon^2\Lambda_{ee}^2 - \upsilon^2\nabla^2)\phi_{ee} = \upsilon^2\Lambda_{ee}^2 Q_e \tag{A.7}$$

$$(\frac{\partial^2}{\partial t^2} + 2\upsilon\Lambda_{ei}\frac{\partial}{\partial t} + \upsilon^2\Lambda_{ei}^2 - \upsilon^2\nabla^2)\phi_{ei} = \upsilon^2\Lambda_{ei}^2 Q_e \tag{A.8}$$

The equations in first order form will be (3):

$$\frac{dV_e}{dt} = \frac{1}{\tau_e}(V_e^{rest} + \Delta V_e^{rest} - V_e + \lambda_{Ach}\rho_e\psi_{ee}\Phi_{ee} + \lambda_i\rho_i\psi_{ie}\Phi_{ie}) \tag{A.9}$$

$$\frac{dV_i}{dt} = \frac{1}{\tau_i}(V_i^{rest} - V_i + \lambda_{Ach}\rho_e\psi_{ei}\Phi_{ei} + \lambda_i\rho_i\psi_{ii}\Phi_{ii}) \tag{A.10}$$

$$\frac{d\Phi_{ee}}{dt} = X_{ee} \tag{A.11}$$

$$\frac{dX_{ee}}{dt} = -2\gamma_{ee}X_{ee} - \gamma_{ee}^2\Phi_{ee} + \gamma_{ee}^2(N_{ee}^{\alpha}\phi_{ee} + N_{ee}^{\beta}Q_e + \phi_{ee}^{sc}) \tag{A.12}$$

$$\frac{d\Phi_{ei}}{dt} = X_{ei} \tag{A.13}$$

$$\frac{dX_{ei}}{dt} = -2\gamma_{ei}X_{ei} - \gamma_{ei}^2\Phi_{ei} + \gamma_{ei}^2(N_{ei}^{\alpha}\phi_{ei} + N_{ei}^{\beta}Q_e + \phi_{ei}^{sc}) \tag{A.14}$$

$$\frac{d\Phi_{ie}}{dt} = X_{ie} \tag{A.15}$$

$$\frac{dX_{ie}}{dt} = -2\gamma_{ie}X_{ie} - \gamma_{ie}^2\Phi_{ie} + \gamma_{ie}^2(N_{ie}^{\beta}Q_i + \phi_{ie}^{sc}) \tag{A.16}$$

$$\frac{d\Phi_{ii}}{dt} = X_{ii} \tag{A.17}$$

$$\frac{dX_{ii}}{dt} = -2\gamma_{ii}X_{ii} - \gamma_{ii}^2\Phi_{ii} + \gamma_{ii}^2(N_{ii}^{\beta}Q_i + \phi_{ii}^{sc}) \tag{A.18}$$

$$\frac{d\phi_{ee}}{dt} = Y_{ee} \tag{A.19}$$

$$\frac{dY_{ee}}{dt} = -2\upsilon\Lambda_{ee}Y_{ee} - \upsilon^2(\Lambda_{ee}^2 - \nabla^2)\phi_{ee} + \upsilon^2\Lambda_{ee}^2Q_e \tag{A.20}$$

$$\frac{d\phi_{ei}}{dt} = Y_{ei} \tag{A.21}$$

$$\frac{dY_{ei}}{dt} = -2\upsilon\Lambda_{ei}Y_{ei} - \upsilon^2(\Lambda_{ei}^2 - \nabla^2)\phi_{ei} + \upsilon^2\Lambda_{ei}^2Q_i \tag{A.22}$$

The equations above can be inputted to the function.
e.g. the 1^{st}-order derivative for V_e appeared in the MATLAB can be:

```
incVe = (1/tau_e)*(Ve_rest+del_Vrest - Ve + rho_e * lambda_Ach * psi_ee.
* PHI_ee + rho_i*lambda_i*psi_ie.*PHI_ie);
```

The equations stored in this function must be modified for changing the model.

A.6.2 `function find_steady_states` (find_steady_states.m)

This function calculates the steady-states of the variables.
The procedure of this function is:

1. Read in the constants and parameters
2. Solve for the equilibrium value of the variables
3. Return the stationary states

This function generates the steady-states for the variables. However, the steady-states of variables V_e, V_i, Q_e and Q_i are defined by `function steady_states_delV_lambda` for modified Liley model. This function loads the equilibrium value of V_e, V_i, Q_e and Q_i, and solves the steady-states for other variables. The definitions for the equilibrium value of the variables (except V_e, V_i, Q_e and Q_i) stored in this function are derived by setting all the differential terms to zero, and they are used for the Steyn-Ross model:

$$\phi_{ee}^{eqm} = Q_e^{eqm} \tag{A.23}$$

$$\phi_{ei}^{eqm} = Q_e^{eqm} \tag{A.24}$$

$$\Phi_{ee}^{eqm} = N_{ee}^{\alpha}\phi_{ee}^{eqm} + N_{ee}^{\beta}Q_e^{eqm} + \phi_{ee}^{sc} \tag{A.25}$$

$$\Phi_{ie}^{eqm} = N_{ie}^{\beta}Q_i^{eqm} + \phi_{ie}^{sc} \tag{A.26}$$

$$\Phi_{ei}^{eqm} = N_{ei}^{\alpha}\phi_{ei}^{eqm} + N_{ei}^{\beta}Q_e^{eqm} + \phi_{ei}^{sc} \tag{A.27}$$

$$\Phi_{ii}^{eqm} = N_{ii}^{\beta}Q_i^{eqm} + \phi_{ii}^{sc} \tag{A.28}$$

Subsidiary variables:

$$X_{ee}^{eqm} = 0 \tag{A.29}$$

$$X_{ei}^{eqm} = 0 \tag{A.30}$$

$$X_{ie}^{eqm} = 0 \tag{A.31}$$

$$X_{ii}^{eqm} = 0 \tag{A.32}$$

$$Y_{ee}^{eqm} = 0 \tag{A.33}$$

$$Y_{ei}^{eqm} = 0 \tag{A.34}$$

The definition of the steady-states for the variables must be written by the user. The initial condition of the variables will be outputed as an `Nspace`×`Nspace` matrix, and all the matrixes will be stored in the parameter f for output.

Testing

This function has a section of testing code in it. This is for triggering the test mode for testing the `function Steady_States_Stability_14`×`14`. The testing parameter value can be changed here by the user or add new case of `del_Vrest` & `lambda_Ach`. These testing parameters will be passed to `function Steady_States_Stability_14x14`. To trigger this test mode, set `O.stability_testing_mode` from 0 to 1, 2 or 3, then type **find_steady_states** in the Command Window.

```
448 - global O
449 - O.stability_testing_mode = 0;  % 1, 2 or 3 are testing mode, 0 is normal mode.
450 - if O.stability_testing_mode ~= 0
451 -     warning('testing mode activated!!! (function find_steady_states)');
452 -     fprintf('\n press any key to proceed testing or Ctrl-C to stop the program \n\n\n');
453 -     pause
454       % testing parameters are for changing here.
455 -     switch O.stability_testing_mode
456 -         case 1
457 -             [O.del_Vrest O.lambda_Ach] = deal(-1.8, 1.15)
458 -             fprintf('\n This should result three unstable states \n\n\n');
459 -         case 2
460 -             [O.del_Vrest O.lambda_Ach] = deal(10, 0.20)  % all stable
```

Testing mode 1 `del_Vrest = -1.8` & `lambda_Ach = 1.15`

This should result three unstable states.

Testing mode 2 `del_Vrest = 10` & `lambda_Ach = 0.2`

This should result all stable.

Testing mode 3 `del_Vrest = 10` & `lambda_Ach = 0.21`

This should result one unstable state.

The other testing parameters can be set as following:

```
lambda_i = 1
gamma_ee = 1342
gamma_ii = 14.53
v = 1400
lamda_ee = 0.20
q = 0
N_ei_a_boost = 1
graphFlag = 1
Nsearch = 20000
Branch = 0
Nspace = 32
```

The parameters above are for the Steyn-Ross implementation of the modified Liley model.

A.6.3 `function steady_states_delV_lambda` (steady_states_delV_lambda.m)

This is a sub-function of `function find_steady_states` and is used in the particular implementation of the stationary states solution of the modified Liley model. However, the user does not need to follow this structure. This function is designed to solve for the steady-states of Q_e, Q_i, V_e and V_i for the modified Liley model with the user-specified input parameters (ΔV_e^{rest}, λ_{Ach} and λ_i). This function was written by D.A. Steyn-Ross. Figure A.6 (Sketched by D.A. Steyn-Ross) shows the

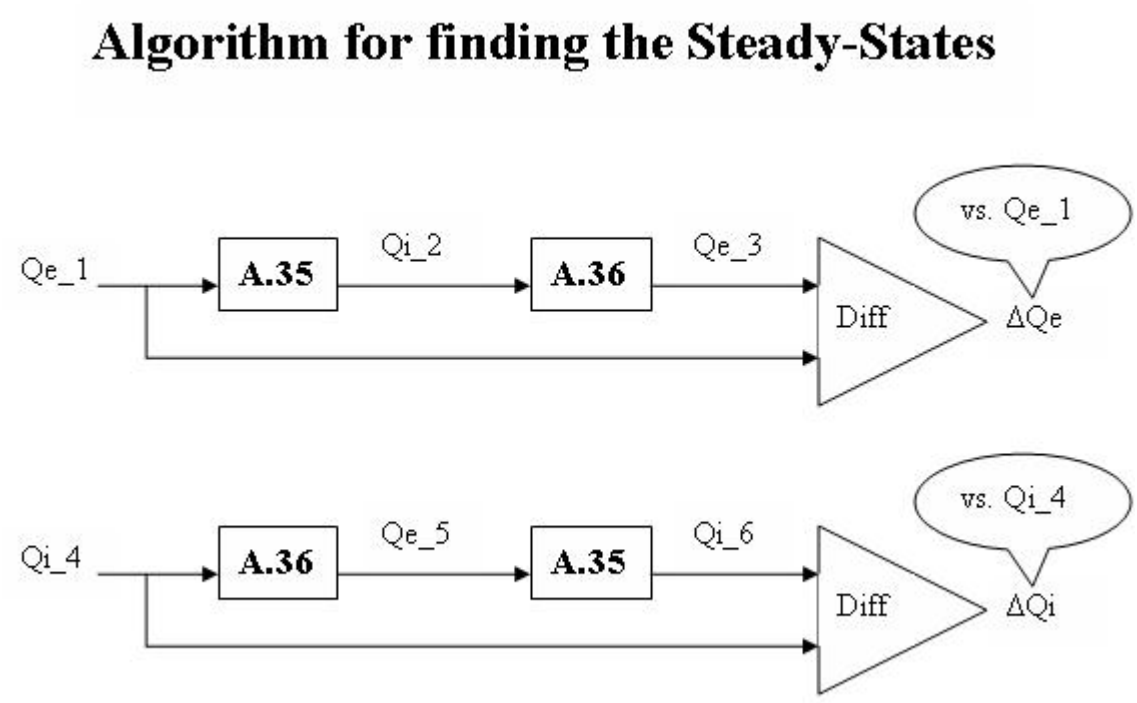


Figure A.6: *algorithm for finding the steady-states*

algorithm for finding the steady-states: The basic idea for finding the steady-state is by guessing and testing. We first note that equations (Equation A.3 - A.8) can be solved very simply to give the equilibrium values of Φ_{ee} etc in terms of Q_e and Q_i. This leaves just two equations (Equation A.1) and (Equation A.2). Therefore we can proceed as follows:

1. *A vector contains a range of values is guessed for Q_e (Qe_1)*

   The range will be between [0 - offset] and [maximum Q_e - offset], where the value for the offset is for the user to define.

2. *Input the guessed Q_e values (Qe_1) to the Equation A.35 to get the corresponding values for Q_i (Qi_2)*

   The equation for A.35 is:

$$Q_i = \frac{1}{N_{ie}^{\beta}}[\frac{V_e - V_e^{rest} - \Delta V_e^{rest} - \lambda_{Ach}\rho_e\psi_{ee}(N_{ee}^{\alpha\beta}Q_e + \phi_{ee}^{sc})}{\rho_i\lambda_i\psi_{ie}} - \phi_{ie}^{sc}] \quad \text{(A.35)}$$

3. *Input these Q_i values (Qi_2) to Equation A.36 to get the corresponding analytical values for Q_e (Qe_3)*

The equation for A.36 is:

$$Q_e = \frac{1}{N_{ei}^{\alpha\beta}}[\frac{V_i - V_i^{rest} - \rho_i\lambda_i\psi_{ii}(N_{ii}^{\beta}Q_i + \phi_{ii}^{sc})}{\lambda_{Ach}\rho_e\psi_{ei}} - \phi_{ei}^{sc}] \tag{A.36}$$

4. *Compare the analytical and guessed Q_e and find the difference between them (ΔQ_e)*

   ΔQ_e = Qe_3 - Qe_1

5. *Define the pair of values for ΔQ_e, which is the closest to zero*

   Since the guessed Q_e is a vector contains a range of values, so the returned ΔQ_e is also a vector and two of ΔQ_e values are the closest to zero, one value is positive and another one is negative (three pairs of ΔQ_e if there are three steady-states).

6. *Define the corresponding Q_e pair from the guessed Q_e vector*

   Once the pair of ΔQ_e is defined, then it can be used to define the corresponding pair in the guessed Q_e vector, because the actual equilibrium for Q_e will be within the range of the difference between both values in the pair.

7. *Use MATLAB built-in function* `fzero` *to find the equilibrium for Q_e*

   Once the corresponding Q_e pair is defined, then it will be input to the MATLAB build-in function `fzero` to locate the value for the equilibrium.

The steps above are the procedure for finding the steady-states for Q_e. The same way applies for finding Q_i as well. Once the steady-states for Q_e and Q_i are defined, then they can be used to define the steady-states for V_e and V_i. This can be done by inputting the Q value into the equation. The equations for $Q_e(V_e)$ and $Q_i(V_i)$ are:

$$Q_e(V_e) = \frac{Q_e^{max}}{1 + exp[-\pi(V_e - \theta_e)/\sqrt{3}\sigma_e]} \tag{A.37}$$

$$Q_i(V_i) = \frac{Q_i^{max}}{1 + exp[-\pi(V_i - \theta_i)/\sqrt{3}\sigma_i]} \tag{A.38}$$

Once the steady-states for variables V_e, V_i, Q_e and Q_i are all defined, then they will be returned by the function.

The equations A.35, A.36 A.37 and A.38 are based on modified Liley model. Table A.1 shows the sub-functions of `function steady_states_delV_lambda` that they store equations and must be modified if the model is changed.

Table A.1: Relationship between the equation and the sub-function of `function steady_states_delV_lambda`

Equation A.35	`function Qi_2 = Eq333`
Equation A.36	`function Qe_3 = Eq334`
$Q_e(V_e)$	`function Qe = Qsige`
$Q_i(V_i)$	`function Qi = Qsigi`
$\frac{dQ_e}{dV_e}$	`function d_Qe = d_Qsige`
$\frac{dQ_i}{dV_i}$	`function d_Qi = d_Qsigi`
$V_e(Q_e)$	`function invsig = invQsige`
$V_i(Q_i)$	`function invsig = invQsigi`
ψ_{ee}	`function weight = Psi_ee`
ψ_{ei}	`function weight = Psi_ei`
ψ_{ie}	`function weight = Psi_ie`
ψ_{ii}	`function weight = Psi_ii`

A.6.4 `function Steady_States_stability_14x14` (Steady_States_stability_14x14.m)

This function is for defining the stability of the stationary states. The procedure to define the stability is:

1. Reads in the parameters and equilibriums
2. Defines the Jacobian matrix
3. Outputs the Eigenvalues, Largest Eigenvalues and Oscillation Frequency

To define the Jacobian matrix, the user must input the partial derivatives of the model's ordinary equations. For example, if the linearized equations of the model are in the form $\frac{dx}{dt} = f(x,t)$ and the equilibrium is the vector x_0, then the partial derivative can be defined by differentiating the linearized equations by the variables. The partial derivative will be in the form $(\frac{\partial f_i}{\partial x_j})_{x_0}$. If there are 14 variables, then there will be 14×14 partial derivatives. The definition of the partial derivatives must be modified for changing the model.

If the real parts of the eigenvalues are all negative, then the stationary states are stable.

Testing Mode

This function has the testing mode in it. The purpose of this is for user to examine the output, to ensure that this it is working properly. This function needs the steady-states before the testing mode starts, so the testing mode of this function is triggered by `function find_steady_states`. The testing parameters are set in

`function find_steady_states`, and it works out the steady-states for the testing cases. Then these testing parameters and steady-states are passed to the stability function to calculate the eigenvalues. The returned eigenvalues should be as the user expected if the stability function is working properly.

A.7 Output (`function data_output` & `function save_output`)

There are two output functions for the system, `function data_output` & `function save_output`. The way they work is that `function data_output` passes the data of the current time-step to the global. `Function save_output` will be called by the system to save all the data with every time-step after the calculations / simulation finished. Therefore, to modify the saving section of the system:

1. Modify the content under the **if** statement (`dataFlag`) in `function data_output` (data_output.m)

```
105 - if (mod(t,t_step_4_tplot) == 0) % recording the svector for every
106 -     tplot = floor(t/t_step_4_tplot)+1
107 -     [dataFlag t_step_4_graph saved_variable] = deal(I.dataFlag, I.
108 -     if (dataFlag == true)
109           % records data to global
110 -         O.svec(:,:,:,tplot) = f(:,:,saved_variable);
111 -         O.Qe(:,:,tplot) = Qsige(f(:,:,1));
112 -         O.Qi(:,:,tplot) = Qsigi(f(:,:,2));
113 -         if (I.stabilityFlag == true)
114               % finds steady-states with current parameters(e.g. lam
115               % and calculates for stability. This is only necessary
116               % is a trajectory.
117 -             find_steady_states;
118 -         end     % end of the stability test
119 -     end         % end of the data calculation and transfering
```

2. Modify the content under the **if** statement (`dataFlag`) in `function save_output` (save_output.m)

```
164 - if (dataFlag == true)
165       % calls out variables from global O & I
166 -     [svec Qe Qi Equilibrium eigenvalues_14x14 largest_eigenvalue_14x14 ...
167           oscillation_freq_14x14] = deal(O.svec, O.Qe, O.Qi, O.Equilibrium, ...
168           O.eigenvalues_14x14, O.largest_eigenvalue_14x14, O.oscillation_freq_14x14);
169 -     [tend Nspace deltat deltax lcortex t_step_4_tplot] = deal(...
170           I.tend, I.Nspace, I.deltat, I.deltax, I.lcortex, I.t_step_4_tplot);
171       % saves the data with stability test to file "firing_rates_etc"
172 -     save(I.DataFileName, ...
173           'svec', 'Qe', 'Qi', 'tend', 'Nspace', 'deltat', 'deltax', 'lcortex', 't_step_4_tplot', 'Equilibrium', ...
174           'eigenvalues_14x14', 'largest_eigenvalue_14x14', 'oscillation_freq_14x14'); % Saves the time and space deta
175 - end
```

A.8 Data Processing for Analysis

This system contains two functions for processing the variable data. They are `function Power_Spectrum` and `function Variable_Variation`. These two functions can only process one variable each time. The user can select the variable for processing by setting the control-flag `I.Analyzed_Variable` in the user input file to the index of the saved variable. E.g. if the index of the variable V_e is the first in `svec` (`svec(:,:,1,:)`), then `I.Analyzed_Variable = 1` for processing the variable V_e.

A.8.1 `function Power_Spectrum`

1. Set the time length for division period and settling-time in the input file (`function I = Input`).

   Division period The whole simulation data can be divided into several pieces by an user-defined time-length, so the function can process the data piece by piece. This is useful for observing any obvious change of the cortical signal in the simulation.
   E.g. set the division period to 2.5 seconds
   `I.division_period = 2.5;`

   Settling time The signal may exhibit transient fluctuations early in the simulation (i.e. not have **settled**). Therefore, the user supplies a time period known as the **settling time** in which the data will not be analyzed and the data within the settling time is going to be taken out from the loaded data.
   E.g. set the settling time to 5 seconds
   `I.settling_time = 5;`

2. Type **Power_Spectrum** in the Command Window to run this function.

   The function should now generate two plots: plot of power vs. temporal frequency and plot of power vs. temporal & spatial frequencies (Figure A.7). This function cuts the data into pieces with user defined division period, and then processes each division separately. The plots in Figure A.7 are useful for defining the property of the cortical signal in time and space. The order of the division being processed is showed on the title of the graphs.

   To change the angle of viewing to the graph:

   (a) Left click on Rotate3D.

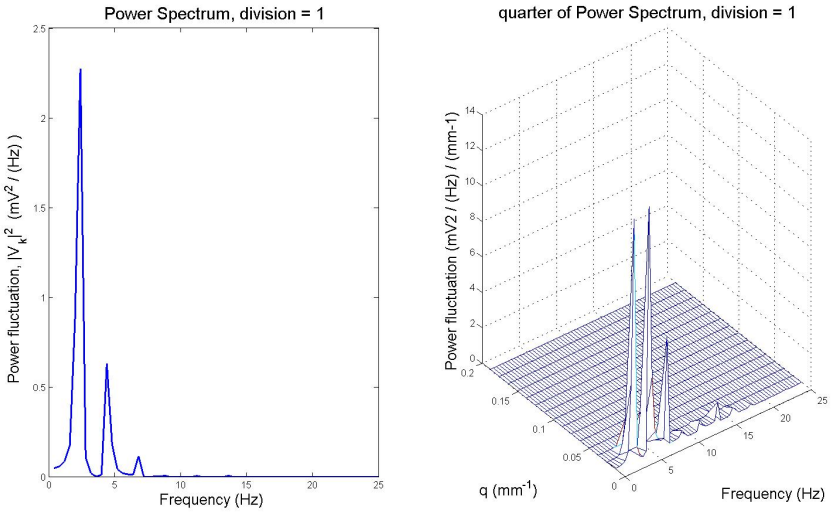
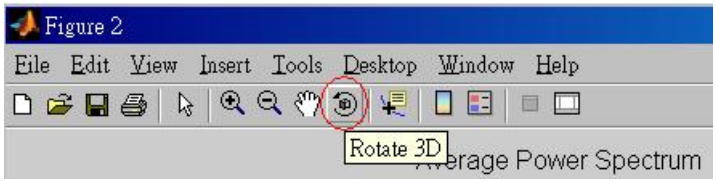


Figure A.7: *Example of Power Spectrum Plot*



(b) Put the mouse cursor on the graph.

(c) Left click and hold to drag the graph for viewing.

3. If there are more than one division, then the function will be paused after processing each division, so press any key when the function is paused.

4. If there are more than one division, then the function will generate the spectrogram plot(Figure A.8). This plot is useful for observing the variation of the cortical signal frequency during the simulation. The other two plots are the averaging power spectrum

Testing the code

This function has built-in testing mode. The purpose of this is for testing the code, to make sure the code is working properly. To test the code:

1. open the file **Power_Spectrum.m**

2. change the value of `testing_mode` to 1, 2 or 3.

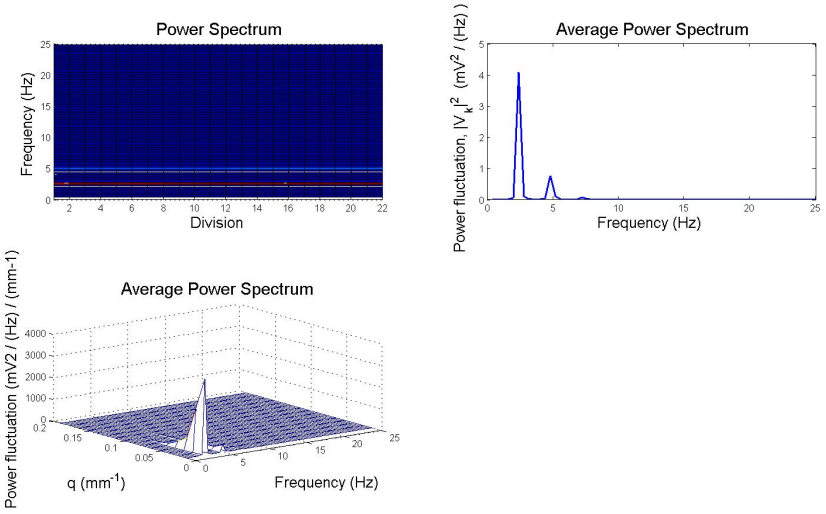
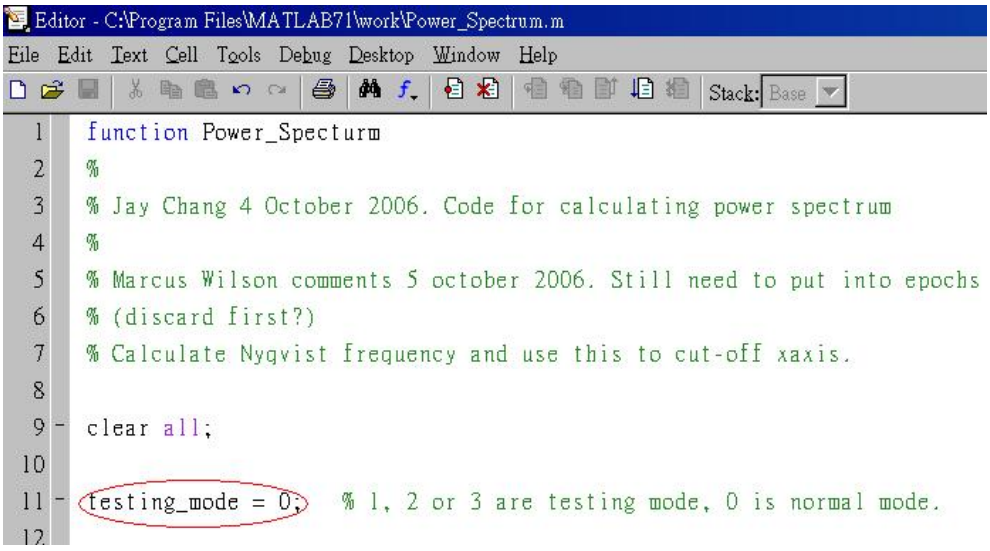


Figure A.8: *Example of Spectrogram plot*



Testing mode 0 The testing mode is not activated. This is the default mode for the function. The function loads and processes the data generated by the main code (`function Cortex_System`).

Testing mode 1 The testing mode is activated. This mode tests the code with a constant frequency. The testing frequency can be changed by changing the value of `testing_frequency` in case 1.

```
35 - switch testing_mode
36
37 -     case 1
38          % This case gives a sine wave signal with constant frequency.
39 -        testing_frequency = 20; % unit in Hz
40 -        xy = 10*sin(2*pi*testing_frequency*testing_time);
41 -        for i = 1:length(testing_time)
42 -            svec(1:Nspace,1:Nspace,1,i) = xy(i);
43 -        end
```

Testing mode 2 The testing mode is activated. This mode tests the code by a signal with frequency proportional to the time. It should result that the peak of the power moving to a higher temporal frequency on the power spectrum plots.

Testing mode 3 The testing mode is activated. This mode tests the code by a signal with frequency proportional to the time, and also some fluctuations in space. It should result that the peak of the power moving to a higher temporal frequency with some ripples in spatial frequency on the power spectrum plots.

The user can modify the existed testing cases or add new testing case to the function in **Power_Spectrum.m**.

A.8.2 `function Variable_Variation`

This function processes the variable data and generates the plots (Figure A.9). The plots are for observing the variation of the cortical signal in time and space.

The x (`user_defined_point_x`) and y (`user_defined_point_y`) coordinates should be specified in the user-input file (`function I = Input`) for processing.
(The value required for x and y coordinates is an integer between 1 to `Nspace`. The actual position (unit in metres) on the cortex plane can be calculated by multiplying this integer by `deltax`.)

For the black & white plot, the whiteness is proportional to the potential. Higher potential results whiter colour and lower potential results blacker colour. This plot is generated by processing the variable data along x axis with a fixed y coordinate (`user_defined_point_y`).

The plot with colour line shows the strength of the cortical signal against time. This is generated by processing the variable data with a fixed (user-defined) point in x and y. The fluctuation can be seen in this plot.

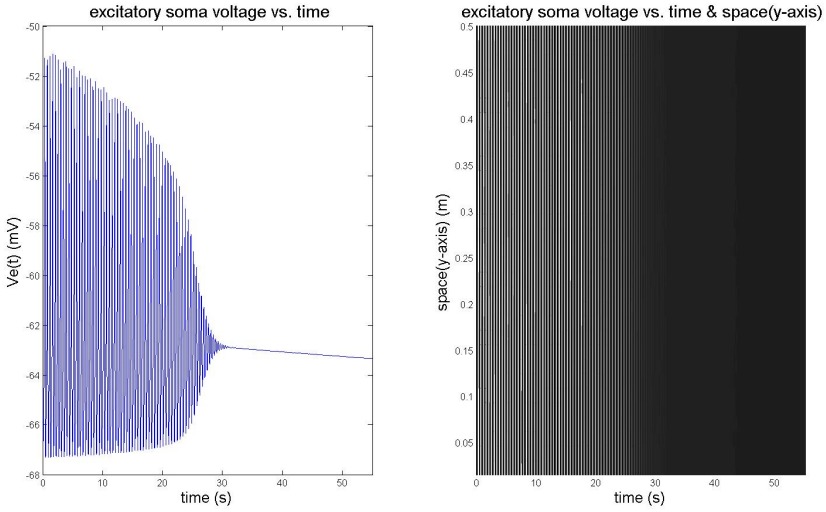


Figure A.9: *Example for the plot of signal variation in time and space*

Appendix B: User Requirement

Requirement	**From**	**Priority**
Must run on PC and MAC	Marcus	High
Must be easy for a user to modify. By 'easy' I mean that equations, parameters, options etc should be kept separate, and organized in a logical way.	Marcus	Essential
Full documentation, including a user-guide, must be written. (This is not necessarily the same as the thesis).	Marcus	Essential
User must be able to supply any model that can be written. as a set of first-order differential equations in the form $dy = f(y)dt + \xi$, where y is a multidimensional vector, f is a function, t is time, and ξ is a noise-input.	Marcus	Essential
Up to 50 first order equations possible (but see next note)	Marcus	High
Model should work in at least two spatial dimensions i.e. so we can implement equations where each component of y is itself a matrix in space.	Marcus	Essential
User should be able to change the equations (model)	Marcus	Essential
User should be able to change values of any parameters	Marcus	Essential
User should be able to specify how different parameters change with time (i.e. do sleep modelling, anaesthesia modelling etc)	Marcus	Essential
User should be able to specify a perturbation in the system at some point(s) in space and time (e.g. k-complex modelling).	Marcus	High

Software should be able to output, at the user's request: • A data-file containting $y_j(r,t)$ where j is any user-specified component of y, r is space and t is time • Movie file of $y_j(r,t)$ in a standard format (e.g. *.avi) • Plots of $y_j(r_a,t)$ where j is any user-specified component of y and r_a is some user-specified point in space • A power spectrum (power against temporal frequency) for the system • A spectrogram plot (power against frequency and time) • Plots of power against temporal frequency and spatial frequency • Stationary (equilibrium) state(s) of the system , and their stability, as a function of time	Marcus	 Essential Medium Medium High Medium Medium High
No third-party software required for which the University of Waikato does not already possess (and intends to keep renewing) a license.	Marcus	High
User should not need to change the main code in order to run the software	Marcus	High
Should be easy to modify the code to use different numerical integration methods (i.e. $2^{nd}/4^{th}$ order RK etc).	Marcus	High
No restrictions on the use of the software by the University of Waikato for academic research purposes.	Marcus	Essential
Ensure, where feasible, that all numerical routines can be tested against standard **calibration** suite.	Alistair	Medium

Appendix C: MATLAB code

C.1 Cortex_System.m

```
%<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< Main function >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
function Cortex_System
%This program looks at the eigth order equation
%Marcus Wilson 30 July 2004
%This programme calls Alistair's to do a loop around the manifold
%11 August 2004  Peforms a simple simulation of the eighth-order system
%12 August 2004 - Let's put some space terms in. 16 spatial points, cyclic
%18 August 2004 - Make it a 2d system, 16 by 16 points.
%19 August 2004 - Puts the subcortical terms in a 'noise' form. See p357
%6 September 2004 - Allow us to move up and down a path in lambda_Ach etc
%22 October 2004 - Does a spatial fourier analysis of the spatial states
%                 and saves the final svec containing the history
%12 November 2004 - See p436. Correction to noise term so it scales the
%spatial noise correctly.
%5 January 2005. Use predictor corrector method. See p 524.
%Theory is in Kloeden and Platen section 15.5
%14 January 2005. Modified to allow us to simulate an anaesthetic. See p536
% Use secs and mm and mV as fundamental units
%24 January 2005. Allows us to find stationary states with lambda_i as a
%control variable
%31 March 2005. delV_rest is a matrix so we can make easy changes to
%excitability. Path trajectories defined.
%

% Jay Chang
% 06 Sep 2006 -
% allow user to select either upper-branch or bottom-branch of the
% equilibrium for the initial condition of the variables.
% 11 Sep 2006 -
% move the derivative part to the first-order derivative function for the
% 14 variables(Ve, Vi, ..., etc.), then change the iterate_by_step function
% to Runge_Kutta_Integration function, and add RK4 method to the function.
% User now can select either RK2 or RK4 for numerical integration.
% 13 Sep 2006 -
% move the section of codes for finding the initial values for the
% variables to steady_states_finder function.
% 14 Sep 2006 -
```

```
% move the section of codes for noises to noise function.
% 20 Sep 2006 -
% move the section of codes for trajectory to find_trajec function.
% move the section of codes for kick change to perturbation function.
% modify the inputs for each input function, so the input functions read
% most of their input parameters from the input function.
% modify the output of noise function: the output of noise function
% overwirtes the noise value into input function.
% modify the output of find_trajec function: the output parameters are
% overwrote into the input function.
% 26 Sep 2006 -
% moved the section of code for plotting the graph to function
% graph_output.
% moved the section of code for saving the movie & data to function
% save_output.
% add flag control for user's option to save the movie & data or not.
% Nov 2006
% add 14x14 & 2x2 stability test functions. data_output function modified.
% the data_output function now can do & record the 14x14 stability test.
clear all;

format long;

global H;
H=init_hasselmo_globs; % Set the global parameters for constants
global I;
I = Input;              % Set the global parameters for inputs

[deltat tend RK Input_FUN] = deal(I.deltat, I.tend, I.RK, I.FUN);

rand('state',57972);        % Generates a random seed

f = find_steady_states;% Now get the initial condition for numerical integration

 for t=0:tend           % Now loop over time

     find_trajec(t);    % Now calculate the trajectory we take in phase space if we need it

     noise;             % Now generate the noise

                        % Now start the incrementation.
     f = Runge_Kutta_Integration(Input_FUN, t, f, deltat, RK);

     perturbation(t);   % Now we do a kick if necessary

     data_output(t, f); % Plot the graph & records data

 end                    % Next time step in the overall loop

    save_output;        % Save the output values & movie
```

```
    fprintf('\nCopyright (C) 2007. The University of Waikato. \nAll Rights Reserved. \n\n http://phys.w

return
%<<<<<<<<<<<<<<<<<<<<<<<<< End of Main function >>>>>>>>>>>>>>>>>>>>>>>>>
```

C.2 Runge_Kutta_Integration.m

```
function f_out = Runge_Kutta_Integration(FUN, time, f, deltat, RK)
% This function does Runge Kutta Integration for the variables. User can
% choose either RK2 or RK4 method for integration.
switch RK
    case 2
        K1 = feval(FUN, time, f, deltat);
        K2 = feval(FUN, (time + deltat), (f + deltat*K1), deltat);
        f_out = f + deltat*(K1 + K2)/2;
    case 4
        K1 = feval(FUN, time, f, deltat);
        K2 = feval(FUN, (time + 0.5*deltat), (f + 0.5*deltat*K1), deltat);
        K3 = feval(FUN, (time + 0.5*deltat), (f + 0.5*deltat*K2), deltat);
        K4 = feval(FUN, (time + deltat), (f + deltat*K3), deltat);
        f_out = f + deltat*(K1 + 2*K2 + 2*K3 + K4)/6;
    otherwise
        warning('Unknown Numerical Integration method!');
        error('Only "2" or "4" are acceptable for "RK" input');
end
return
```

C.3 Input.m

```
function I = Input

I.Nspace = 32;      %Number of discrete points

% input parameters for main function
I.deltat = 0.2*1e-3;%Time step (first part in ms)
I.tend = 1000;       % Total time steps for simulation.
I.RK = 2;           % Runge Kutta Integration, 2 = RK2 ; 4 = RK4
I.FUN = 'first_order_derivative';   % name of the function with 1st order derivative eqns for variables

% input parameters for Liley's model
I.del_Vrest = -1.8;     % effect of altering extrasynaptic ion channels. (unit: mV)
I.lambda_Ach = 1.25; % effect of muscarinic neuromodulation
I.lambda_i = 1;        %An INITIAL value of lambda_i - WILL GET OVERWRITTEN later. Lets us find station
I.gamma_i = 65;         % inhibitory synaptic rate constant.
I.gamma_e = 300;        % excitatory synaptic rate constant.
I.lamda_ee = 0.20;      % characteristic inverse length-scale for connections. Lambda in mm-1 - scaled
I.nu = 1;               %How much noise simulation do we do. 1 = pure poisson sc terms, 0 = pure eqm su
I.v = 1400;             % mean axonal conduction speed.
I.q = 0;               %mtw 21 november 2006. Do q=0 initially. Then try other values.
```

```
% input parameters for find_initial_states function
I.branch = 1;       %for selecting starting point from:   0 = Bottom branch  or   1 = Upper Branch
I.graphFlag = 0;    % plot graphs of steady-state error residuals. 1 = enable graph plotting    ;   0 = disab

% input parameters for find_trajec function
I.trajectory = 0;   %Flag for trajectory - do we have a drug trajectory (1) a user-path (2) or no path (0)

% input parameters for noise function
I.poissflag = 0;    %do we want the poisson noise? THINK CAREFULLY. 1=poisson, 0=gaussian imitation

% input parameters for perturbation function
I.kicker = 0;   % Perturbation flag:    choose true to activate perturbation or false to deactivate
I.kick_division_period = 1; % division period for perturbation
I.kickTime = 0.1;   % time duration of kicking
I.kickVoltage = 10; % Apply 10mV kick
I.x_area = 16;%12:13;   % area for applying kick
I.y_area = 16;%16:17;

% input parameters for grapg_output function
I.t_step_4_tplot = 10;      % user defined No. of t steps for tplot increment
I.t_step_4_graph = 100;     % user defined No. of t steps for plot (No graph will be generated if set to 0, i
I.plotted_variable = 1;     % user defined varaible for plot (1 is Ve)
I.saved_variable = 1;     % user defined varaible to be saved (1 is Ve, [2 4 6] means Vi, PHI_ei & PHI_ii are
I.x_axis_range = [1 I.Nspace];
I.y_axis_range = [1 I.Nspace];
I.z_axis_range = [-70 -35];
I.xtick = [0, 0.25, 0.5 0.75, 1.0]*I.Nspace;
I.ytick = [0, 0.25, 0.5 0.75, 1.0]*I.Nspace;
I.x_axis_label = 'number';
I.y_axis_label = '';
I.z_axis_label = 'Ve /mV';

% input parameters for save_output function
I.dataFlag = 0;         % true (or 1) = enable data saving ;   false (or 0) = disable
I.stabilityFlag = 0; % true (or 1) = do stability test  ;   false (or 0) = ignore stability test
I.movieFlag = 0;        % true (or 1) = saving the movie   ;   false (or 0) = don't save the movie
I.DataFileName = 'firing_rates_etc';    % file name for the data output
I.MovieFileName = 'movie_temp';         % file name for the movie output

% the following empty are created for data saving, do NOT delete or modify
global O
O.Equilibrium = [];             % empty array for stationary states results
O.eigenvalues_14x14 = [];       % empty array for eigenvalues results
O.largest_eigenvalue_14x14 = [];% empty array for largest eigenvalues results
O.oscillation_freq_14x14 = [];  % empty array for oscillation frequency results

%*************************************************************************
% user input parameters for both function power_spectrum and function
% Variable_Variation
```

```
I.Analyzed_Variable = 1;
I.settling_time = 5;    % the data within the settling time will be ignored for processing (unit in sec

% user input parameters for processing power spectrum (function
% Power_Specturm)
I.division_period = 2.5;% user-defined value for fourier transform time interval (unit in second)

% user input parameters for plotting soma potential (function Variable_Variation)
I.user_defined_point_x = 8;
I.user_defined_point_y = 16;

%************************************************************************
% DO NOT modify the parameters below unless necessary...
I.lcortex = 0.5*1e3;% Cortex is 50 cm in size. We use mm in calculation
I.deltax = I.lcortex/I.Nspace;          % Step size
areapermac = 1^2;   %Area per macrocolumn in mm2. NOTE UNITS Usually 1^2 mm2
I.macsperbrain = I.lcortex^2 / areapermac;  %Number of macrocolumns in our simulation
I.del_Vrest_mat = I.del_Vrest*(zeros(I.Nspace,I.Nspace)+1);  %Delta V rest in matrix form
I.Nsearch = 20000;
I.gamma_inhib_standard = I.gamma_i;     %The standard value of gamma_i which the drug application scale
I.lambda_inhib_standard = I.lambda_i; %The standard value of lambda_i which the drug scales.

return
```

C.4 perturbation.m

```
function perturbation(t)
% This function apply a perturbation if the user want to kick change
global I;
if I.kicker == true
    [kickTime kickVoltage x_area y_area deltat del_Vrest ...
        kick_division_period] = deal(I.kickTime, I.kickVoltage, ...
        I.x_area, I.y_area, I.deltat, I.del_Vrest, I.kick_division_period);
    % How frequently and for how long do we apply a kick for. 10 mV for 0.1
    % s generally good
    if (mod(t*deltat-1,kick_division_period) < kickTime)
        %Apply kick to area
        I.del_Vrest_mat(x_area,y_area) = del_Vrest + kickVoltage;
    else
        %remove kick
        I.del_Vrest_mat(x_area,y_area) = del_Vrest;
    end
end
return
```

C.5 noise.m

```
function noise
% This function generates the noise for the cortex
     %Poisson for noise on subcortical terms. Do before predictor because
```

```
     %we need to keep same random term for both halves.
     %CORRECTION 12 Nov 04 - p476 and p434
% 01 Feb 2007 - modified to use FOR loop to replace the repeating codes
global I;
[Nspace macsperbrain deltat poissflag] = ...
    deal(I.Nspace, I.macsperbrain, I.deltat, I.poissflag);

global H;
% calles the mean firing per sec per neuron
phi_sc = [H.phi_ee_sc H.phi_ei_sc H.phi_ie_sc H.phi_ii_sc];

vmat=[Nspace Nspace];  %defines size of matrix of noise terms

%Poisson lambda for the timestep deltat - i.e. expected number of pulses
lambda_phi = macsperbrain*phi_sc * deltat/Nspace^2;

for i = 1 : length(phi_sc)
    if (poissflag)
        %Generates the random matrix for the phi_sc and return the noise
        %vectors to global
        I.R(:,:,i) = poissrnd(lambda_phi(i), vmat);
    else
        %Generates gaussian to speed things up
        I.R(:,:,i) = normrnd(lambda_phi(i), sqrt(lambda_phi(i)), vmat);
    end
end

return
```

C.6 find_trajec.m

```
function find_trajec(t)
%This function does a trajectory in phase space for a given time
%Marcus Wilson 14 January 2005
% See p536 in notebook
% 20 March 2005 - Changed to be more accurate - as in the track_drugs
% programme.
%
global I;
[tend lambda_i gamma_i lambda_Ach del_Vrest Nspace trajectory ...
    gamma_inhib_standard lambda_inhib_standard] = ...
    deal(I.tend, I.lambda_i, I.gamma_i, I.lambda_Ach, I.del_Vrest, ...
    I.Nspace, I.trajectory, I.gamma_inhib_standard, I.lambda_inhib_standard);

switch trajectory
    case 0
        I.lambda_i=1;
        %For normal cases lambda_i remains at 1 - equiv of lambda_ACh
        %gamma_i is untouched.
    case 1
```

```
        max_drug=1.2;           %drug in mM end of path
        x=max_drug*(t/tend) ; %Our value for the drug conc
        %enflurane
        gamma_i_ratio = -0.9679*x^3 + 2.7015*x^2 - 2.4783*x + 0.9961;
        lambda_inhib_ratio = -1.2106*x^3 - 0.1915*x^2 + 2.7004*x + 0.9761;

        %isoflurane
%        if (x < 0.6);
%            gamma_i_ratio = -1.9017*x^3 + 4.4384*x^2 - 3.228*x + 0.9878;
%        else
%            gamma_i_ratio = 0.23;
%        end
%        lambda_inhib_ratio = -3.7335*x^3 +2.1785*x^2 + 4.4188*x + 1.0125;

I.gamma_i = gamma_inhib_standard*gamma_i_ratio;         % Whatever for our drug
I.lambda_i = lambda_inhib_standard*lambda_inhib_ratio; % Whatever for our drug

    case 2
            I.lambda_Ach = lambda_Ach - 0.25/tend;

    otherwise
        warning('Unknown trajectory!');
        error('Only "0", "1" or "2" are acceptable for "trajectory" input');
end

return
```

C.7 first_order_derivative.m

```
function dfdt = first_order_derivative(time, f, deltat)
% This function finds the first order derivative of the variables
global H;
global I;
Ve = f(:,:,1);
Vi = f(:,:,2);
PHI_ee = f(:,:,3);
Xee = f(:,:,4);
PHI_ei = f(:,:,5);
Xei = f(:,:,6);
PHI_ie = f(:,:,7);
Xie = f(:,:,8);
PHI_ii = f(:,:,9);
Xii = f(:,:,10);
phi_ee = f(:,:,11);
Yee = f(:,:,12);
phi_ei = f(:,:,13);
Yei = f(:,:,14);

% calls out the user-input figures
[lambda_Ach gamma_e gamma_i nu v lamda_ee ...
```

```
    deltax Nspace macsperbrain lambda_i del_Vrest] = deal( ...
    I.lambda_Ach, I.gamma_e, I.gamma_i, I.nu, I.v, I.lamda_ee, ...
    I.deltax, I.Nspace, I.macsperbrain, I.lambda_i, I.del_Vrest_mat);

% calls out the constants
[Ve_rev Vi_rev Ve_rest Vi_rest rho_i rho_e ...
    N_ee_a N_ei_a N_ee_b N_ei_b N_ie_b N_ii_b ...
    phi_ee_sc phi_ei_sc phi_ie_sc phi_ii_sc ...
    ] = deal(H.Ve_rev, H.Vi_rev, H.Ve_rest, H.Vi_rest, H.gi, H.ge, ...
    H.Nee_a, H.Nei_a, H.Nee_b, H.Nei_b, H.Nie_b, H.Nii_b, ...
    H.phi_ee_sc, H.phi_ei_sc, H.phi_ie_sc, H.phi_ii_sc);

% calls out the noises
[Ree Rei Rie Rii] = deal(I.R(:,:,1), I.R(:,:,2), I.R(:,:,3), I.R(:,:,4));

tau_e = H.tau_e * 0.001; %Puts tau_e in seconds
tau_i = H.tau_i * 0.001; %Puts tau_i in seconds

%Calculate the derivatives - use vector form
psi_ee = Psi_ee(Ve);
psi_ei = Psi_ei(Vi);
psi_ie = Psi_ie(Ve);
psi_ii = Psi_ii(Vi);
Qe=Qsige(Ve);
Qi=Qsigi(Vi);
     %

incVe = (1/tau_e)*(Ve_rest+del_Vrest - Ve + rho_e*lambda_Ach*psi_ee.*...
    PHI_ee + rho_i*lambda_i*psi_ie.*PHI_ie);
incVi = (1/tau_i)*(Vi_rest - Vi + rho_e*lambda_Ach*psi_ei.*PHI_ei + ...
    rho_i*lambda_i*psi_ii.*PHI_ii);
incPHI_ee = Xee;
incXee = -2*gamma_e*Xee - gamma_e^2*PHI_ee + gamma_e^2*(N_ee_a*phi_ee + ...
    N_ee_b*Qe + (1-nu)*phi_ee_sc+nu*Ree*Nspace^2/(deltat*macsperbrain));
incPHI_ie = Xie;
incXie = -2*gamma_i*Xie - gamma_i^2*PHI_ie + gamma_i^2*(N_ie_b*Qi + ...
    (1-nu)*phi_ie_sc+nu*Rie*Nspace^2/(deltat*macsperbrain));
incPHI_ii = Xii;
incXii = -2*gamma_i*Xii - gamma_i^2*PHI_ii + gamma_i^2*(N_ii_b*Qi + ...
    (1-nu)*phi_ii_sc+nu*Rii*Nspace^2/(deltat*macsperbrain));
incPHI_ei = Xei;
incXei = -2*gamma_e*Xei - gamma_e^2*PHI_ei + gamma_e^2*(N_ei_a*phi_ei + ...
    N_ei_b*Qe + (1-nu)*phi_ei_sc+nu*Rei*Nspace^2/(deltat*macsperbrain));
incphi_ee = Yee;
incphi_ei = Yei;
%Next are the space terms. Use cyclic boundary conditions
for x=1:Nspace; %Do the x term
    xup=x+1;
    if (xup > Nspace)
        xup=1;  %cyclic permute
```

```
    end
    xdown=x-1;
    if (xdown < 1)
        xdown=Nspace;
    end

    for y=1:Nspace  %Now do the y term
        yup = y+1;
        if (yup > Nspace)
            yup=1;
        end
        ydown=y-1;
        if (ydown < 1)
            ydown=Nspace;
        end
        incYee(x,y) = -2*v*lamda_ee*Yee(x,y) - lamda_ee^2*v^2*phi_ee(x,y) + ...
            v^2*(-4*phi_ee(x,y)+phi_ee(xup,y) + phi_ee(xdown,y) + ...
            phi_ee(x,yup)+phi_ee(x,ydown))/deltax^2 + v^2*lamda_ee^2*Qe(x,y);
        incYei(x,y) = -2*v*lamda_ee*Yei(x,y) - lamda_ee^2*v^2*phi_ei(x,y) + ...
                    v^2*(-4*phi_ei(x,y)+phi_ei(xup,y)+ phi_ei(xdown,y) + ...
                    phi_ei(x,yup)+phi_ei(x,ydown)) / deltax^2 + ...
                    v^2*lamda_ee^2*Qe(x,y);
                %Now just do the next x and y
    end
end

% return the 1st-order-derivative of the variables
dfdt(:,:,1) = incVe;
dfdt(:,:,2) = incVi;
dfdt(:,:,3) = incPHI_ee;
dfdt(:,:,4) = incXee;
dfdt(:,:,5) = incPHI_ei;
dfdt(:,:,6) = incXei;
dfdt(:,:,7) = incPHI_ie;
dfdt(:,:,8) = incXie;
dfdt(:,:,9) = incPHI_ii;
dfdt(:,:,10) = incXii;
dfdt(:,:,11) = incphi_ee;
dfdt(:,:,12) = incYee;
dfdt(:,:,13) = incphi_ei;
dfdt(:,:,14) = incYei;
return
```

C.8 find_steady_states.m

```
function f = find_steady_states
% This function finds the initial condition for the variables
global H;

%************************************************************************
```

```
% This section is the code of testing mode.

global O
O.stability_testing_mode = 0;   % 1, 2 or 3 are testing mode, 0 is normal mode.
if O.stability_testing_mode ~= 0
    warning('testing mode activated!!! (function find_steady_states)');
    fprintf('\n press any key to proceed testing or Ctrl-C to stop the program \n\n\n');
    pause
    % testing parameters are for changing here.
    switch O.stability_testing_mode
        case 1
            [O.del_Vrest O.lambda_Ach] = deal(-1.8, 1.15)
            fprintf('\n This should result three unstable states \n\n\n');
        case 2
            [O.del_Vrest O.lambda_Ach] = deal(10, 0.20)   % all stable
            fprintf('\n This should result all stable \n\n\n');
        case 3
            [O.del_Vrest O.lambda_Ach] = deal(10, 0.21)   % 1 unstable
            fprintf('\n This should result one unstable state \n\n\n');
        otherwise
            error('Unknown 14x14 Stability testing case!');
    end
    O.lambda_i = 1;
    O.gamma_ee = 1342;
    O.gamma_ii = 14.53;
    [O.v O.lamda_ee O.q O.N_ei_a_boost] = deal(1400, 0.20, 0, 1);

    [del_Vrest lambda_Ach lambda_i] = ...
        deal(O.del_Vrest, O.lambda_Ach, O.lambda_i);
    [graphFlag Nsearch branch Nspace] = ...
        deal(1, 20000, 0, 32);

    H=init_hasselmo_globs; % Set the global parameters for constants
else
%*************************************************************************
    global I;
    % calls out the user input parameters
    [del_Vrest lambda_Ach lambda_i graphFlag Nsearch branch Nspace] = ...
        deal(I.del_Vrest, I.lambda_Ach, I.lambda_i, I.graphFlag, ...
        I.Nsearch, I.branch, I.Nspace);
end

% calls out the constants
[N_ee_a N_ei_a N_ee_b N_ei_b N_ie_b N_ii_b phi_ee_sc phi_ei_sc phi_ie_sc ...
    phi_ii_sc] = deal(H.Nee_a, H.Nei_a, H.Nee_b, H.Nei_b, H.Nie_b, ...
    H.Nii_b, H.phi_ee_sc, H.phi_ei_sc, H.phi_ie_sc, H.phi_ii_sc);

%Call alistairs code to get steady state solution
[Qeeqm, Qieqm, Veeqm, Vieqm]=steady_states_delV_lambda(del_Vrest,...
    lambda_Ach,lambda_i,graphFlag,Nsearch);
```

```
% choose either top or bottom branch for the initial condition
switch branch
    case 0
        [Qe_eqm Qi_eqm Ve_eqm Vi_eqm] = ...
            deal(Qeeqm(1), Qieqm(1), Veeqm(1), Vieqm(1));
    case 1
        [Qe_eqm Qi_eqm Ve_eqm Vi_eqm] = ...
            deal(Qeeqm(end), Qieqm(end), Veeqm(end), Vieqm(end));
    otherwise
        warning('Unknown Branch!');
        error('Only "0" or "1" are acceptable for "Branch" input');
end

%Load-up equilibrium values

phi_ee_eqm = Qe_eqm;
phi_ei_eqm = Qe_eqm;
PHI_ee_eqm = N_ee_a*phi_ee_eqm + N_ee_b*Qe_eqm + phi_ee_sc;
PHI_ie_eqm = N_ie_b*Qi_eqm + phi_ie_sc;
PHI_ei_eqm = N_ei_a*phi_ei_eqm + N_ei_b*Qe_eqm + phi_ei_sc;
PHI_ii_eqm = N_ii_b*Qi_eqm + phi_ii_sc;

%Subsidiary variables
Xee_eqm = 0;
Xei_eqm = 0;
Xie_eqm = 0;
Xii_eqm = 0;
Yee_eqm = 0;
Yei_eqm = 0;

equilibrium = [Ve_eqm; Vi_eqm; PHI_ee_eqm; Xee_eqm; PHI_ei_eqm; Xei_eqm; ...
    PHI_ie_eqm; Xie_eqm; ; PHI_ii_eqm; Xii_eqm; phi_ee_eqm; Yee_eqm; ...
    phi_ei_eqm; Yei_eqm];

[eigenvalues_14x14, largest_eigenvalue_14x14, oscillation_freq_14x14] ...
    = Steady_States_stability_14x14(equilibrium);

if (O.stability_testing_mode == 0) && (I.dataFlag == true)
    global O;   % pass the stability data to global for saving
    O.Equilibrium = [O.Equilibrium, equilibrium];
    O.eigenvalues_14x14 = [O.eigenvalues_14x14, eigenvalues_14x14];
    O.largest_eigenvalue_14x14 = ...
        [O.largest_eigenvalue_14x14, largest_eigenvalue_14x14];
    O.oscillation_freq_14x14 = ...
        [O.oscillation_freq_14x14, oscillation_freq_14x14];
end

%**************************************************************************
% This section is for displaying the steady-states data in the command
```

```
% window (optional, DATA DISPLAY purpose)
if (O.stability_testing_mode ~= 0) || (I.stabilityFlag == false)
    Qeeqm
    Qieqm
    Veeqm
    Vieqm
    switch branch
        case 0
            fprintf('\n Bottom Branch \n\n\n');
        case 1
            fprintf('\n Upper Branch \n\n\n');
    end
    fprintf('\n 14x14 Steady States stability: \n\n');
    eigenvalues_14x14
    largest_eigenvalue_14x14
    oscillation_freq_14x14
    fprintf('\n end of 14x14 Steady States stability test \n\n');
end
%*************************************************************************

%Loop over all space
x=1:Nspace;
y=x;

for i = 1:length(equilibrium)
    f(x,y,i) = equilibrium(i);
end

return
```

C.9 Steady_States_stability_14x14.m

```
function [eigenvalues, largest_eigenvalue, oscillation_freq] ...
    = Steady_States_stability_14x14(equilibrium)

%*************************************************************************
% This section is the code of testing mode.
global O
if O.stability_testing_mode ~= 0
    % calls the testing parameters
    [del_Vrest lambda_Ach lambda_i v gamma_ee gamma_ii lamda_ee q] = ...
        deal(O.del_Vrest, O.lambda_Ach, O.lambda_i, O.v, O.gamma_ee, ...
        O.gamma_ii, O.lamda_ee, O.q);
    fprintf('\n testing mode activated (Steady_States_stability_14x14) \n\n\n');
else
%*************************************************************************
    global I;
    [del_Vrest lambda_Ach lambda_i v gamma_ee gamma_ii lamda_ee q] = ...
        deal(I.del_Vrest, I.lambda_Ach, I.lambda_i, I.v, I.gamma_e, ...
        I.gamma_i, I.lamda_ee, I.q);
```

```
end

[Ve Vi PHI_ee Xee PHI_ei Xei PHI_ie Xie PHI_ii Xii phi_ee Yee phi_ei Yei] ...
    = deal(equilibrium(1), equilibrium(2), equilibrium(3), equilibrium(4), ...
    equilibrium(5), equilibrium(6), equilibrium(7), equilibrium(8), ...
    equilibrium(9), equilibrium(10), equilibrium(11), equilibrium(12), ...
    equilibrium(13), equilibrium(14));

global H;
[Ve_rev Vi_rev Ve_rest Vi_rest rho_i rho_e ...
    N_ee_a N_ei_a N_ee_b N_ei_b N_ie_b N_ii_b] = deal(...
    H.Ve_rev, H.Vi_rev, H.Ve_rest, H.Vi_rest, H.gi, H.ge, ...
    H.Nee_a, H.Nei_a, H.Nee_b, H.Nei_b, H.Nie_b, H.Nii_b);

gamma_ei = gamma_ee;
gamma_ie = gamma_ii;
lamda_ei = lamda_ee;

tau_e = H.tau_e * 0.001; %Puts tau_e in seconds
tau_i = H.tau_i * 0.001; %Puts tau_i in seconds

%Calculate the derivatives - use vector form
psi_ee = Psi_ee(Ve);
psi_ei = Psi_ei(Vi);
psi_ie = Psi_ie(Ve);
psi_ii = Psi_ii(Vi);

% creates a 14x14 matrix for Jacobian and initializes it with zeros
J = zeros(14,14);

% Eqns & Variables sequence for the Jacobian matrix:
% 1:Ve, 2:Vi, 3:PHI_ee, 4:Xee, 5:PHI_ei, 6:Xei, 7:PHI_ie, 8:Xie, 9:PHI_ii,
% 10:Xii, 11:phi_ee, 12:ZYee, 13:phi_ei, 14:Yei

% partial derivatives from Eqs. (A1)
%mtw 21 november 2006 correction. 1/tau_e not tau_e
ddVe_dVe = (1/tau_e)*(-1 - rho_e*lambda_Ach*PHI_ee/(Ve_rev-Ve_rest) - ...
    rho_i*lambda_i*PHI_ie/(Vi_rev-Ve_rest));
ddVe_dPHI_ee = (1/tau_e)*(rho_e*lambda_Ach*psi_ee);
ddVe_dPHI_ie = (1/tau_e)*(rho_i*lambda_i*psi_ie);
J(1,1) = ddVe_dVe;      % puts to Jacobian matrix
J(1,3) = ddVe_dPHI_ee;
J(1,7) = ddVe_dPHI_ie;

% partial derivatives from Eqs. (A2)
%mtw 21 november 2006. Correction. 1/tau_i not tau_i
ddVi_dVi = (1/tau_i)*(-1 - rho_e*lambda_Ach*PHI_ei/(Ve_rev-Vi_rest) - ...
    rho_i*lambda_i*PHI_ii/(Vi_rev-Vi_rest));
ddVi_dPHI_ei = (1/tau_i)*(rho_e*lambda_Ach*psi_ei);
ddVi_dPHI_ii = (1/tau_i)*(rho_i*lambda_i*psi_ii);
```

```
J(2,2) = ddVi_dVi;
J(2,5) = ddVi_dPHI_ei;
J(2,9) = ddVi_dPHI_ii;

% partial derivatives from Eqs. (A3)
ddPHI_ee_dXee = 1;
ddXee_dXee = -2 * gamma_ee;
ddXee_dPHI_ee = - gamma_ee^2;
ddXee_dphi_ee = gamma_ee^2 * N_ee_a;
ddXee_dVe = gamma_ee^2 * N_ee_b * d_Qsige(Ve);
J(3,4) = ddPHI_ee_dXee;
J(4,4) = ddXee_dXee;
J(4,3) = ddXee_dPHI_ee;
J(4,11) = ddXee_dphi_ee;
J(4,1) = ddXee_dVe;

% partial derivatives from Eqs. (A4)
ddPHI_ei_dXei = 1;
ddXei_dXei = -2 * gamma_ei;
ddXei_dPHI_ei = - gamma_ei^2;
ddXei_dphi_ei = gamma_ei^2 * N_ei_a;
ddXei_dVe = gamma_ei^2 * N_ei_b * d_Qsige(Ve);
J(5,6) = ddPHI_ei_dXei;
J(6,6) = ddXei_dXei;
J(6,5) = ddXei_dPHI_ei;
J(6,13) = ddXei_dphi_ei;
J(6,1) = ddXei_dVe;

% partial derivatives from Eqs. (A5)
ddPHI_ie_dXie = 1;
ddXie_dXie = -2 * gamma_ie;
ddXie_dPHI_ie = - gamma_ie^2;
ddXie_dVi = gamma_ie^2 * N_ie_b * d_Qsigi(Vi);
J(7,8) = ddPHI_ie_dXie;
J(8,8) = ddXie_dXie;
J(8,7) = ddXie_dPHI_ie;
J(8,2) = ddXie_dVi;

% partial derivatives from Eqs. (A6)
ddPHI_ii_dXii = 1;
ddXii_dXii = -2 * gamma_ii;
ddXii_dPHI_ii = - gamma_ii^2;
ddXii_dVi = gamma_ii^2 * N_ii_b * d_Qsigi(Vi);
J(9,10) = ddPHI_ii_dXii;
J(10,10) = ddXii_dXii;
J(10,9) = ddXii_dPHI_ii;
J(10,2) = ddXii_dVi;

% partial derivatives from Eqs. (A7)
ddphi_ee_dYee = 1;
```

```
ddYee_dYee = -2*v*lamda_ee;
ddYee_dphi_ee = -v^2*lamda_ee^2 - v^2*q^2;
ddYee_dVe = v^2*lamda_ee^2*d_Qsige(Ve);
J(11,12) = ddphi_ee_dYee;
J(12,12) = ddYee_dYee;
J(12,11) = ddYee_dphi_ee;
J(12,1) = ddYee_dVe;

% partial derivatives from Eqs. (A8)
ddphi_ei_dYei = 1;
ddYei_dYei = -2*v*lamda_ei;
ddYei_dphi_ei = -v^2*lamda_ei^2 - v^2*q^2;
ddYei_dVe = v^2*lamda_ei^2*d_Qsige(Ve);
J(13,14) = ddphi_ei_dYei;
J(14,14) = ddYei_dYei;
J(14,13) = ddYei_dphi_ei;
J(14,1) = ddYei_dVe;

eigenvalues = eig(J);      % evaluates the eigenvalues
[largest_eigenvalue, index] = max(real(eigenvalues));
largest_eigenvalue = eigenvalues(index);    % derive the largest eigenvalue
% finds the oscillation frequency (unit in Hz)
oscillation_freq = imag(largest_eigenvalue) / (2*pi);

return
```

C.10 data_output.m

```
function data_output(t, f)
% This function plots the graph of the chosen variable & output the data
% to global for saving
% 8 Jan 2006 -
% the value for tplot is defined by t & t_step_4_tplot, instead of calling
% from global I.
% 9 Jan 2006 -
% add stability test in this function.
global I;
global O;

t_step_4_tplot = I.t_step_4_tplot;

% recording the svector for every number of t steps
if (mod(t,t_step_4_tplot) == 0)
    tplot = floor(t/t_step_4_tplot)+1;
    [dataFlag t_step_4_graph saved_variable tend] = deal(I.dataFlag, ...
        I.t_step_4_graph, I.saved_variable, I.tend);
    % calculate and display the % of the completion for the simulation
    fprintf('Simulation Progress: %g%% completed.\n\n',t/tend*100)
    if (dataFlag == true)
        % records data to global
```

```
        f(:,:,end+1) = Qsige(f(:,:,1));
        % put Qe into f matrix to be the 15th variable
        f(:,:,end+1) = Qsigi(f(:,:,2));
        % put Qi into f matrix to be the 16th variable
        O.svec(:,:,:,tplot) = f(:,:,saved_variable);
        if (I.stabilityFlag == true)
            % finds steady-states with current parameters(e.g. lambda_Ach)
            % and calculates for stability. This is only necessary if there
            % is a trajectory.
            find_steady_states;
        end     % end of the stability test
    end         % end of the data calculation and transfering

    % Plot out the simulated cortex
    if (t_step_4_graph ~= 0) && (mod(t,t_step_4_graph) == 0)
        t_movie = floor(t/t_step_4_graph)+1;
        [movieFlag plotted_variable x_axis_range y_axis_range ...
            z_axis_range xtick ytick x_axis_label y_axis_label ...
            z_axis_label] = deal(...
            I.movieFlag, I.plotted_variable, I.x_axis_range, ...
            I.y_axis_range, I.z_axis_range, I.xtick, I.ytick, ...
            I.x_axis_label, I.y_axis_label, I.z_axis_label);
        figure(3)   % 3-D cortex
        mesh(f(:,:,plotted_variable))
        %surf(f(:,:,plotted_variable))
        set(gca,'xlim',x_axis_range);
        set(gca,'ylim',y_axis_range);
        set(gca,'zlim',z_axis_range);
        set(gca,'xtick',xtick);
        set(gca,'ytick',ytick);
        xlabel(x_axis_label);
        ylabel(y_axis_label);
        zlabel(z_axis_label);

        drawnow;

        if (movieFlag == true)
            % records the frame for the movie to global
            O.Mmovie(t_movie)=getframe;
        end

        figure(2)   % 2-D cortex
        mesh(f(:,:,plotted_variable))
        %surf(f(:,:,plotted_variable))
        view(0, 90);
        drawnow;

    end % end tplot if statement for meshing
end     % end tplot if statement for recording the svector
```

```
return
```

C.11 save_output.m

```
function save_output
% This function saves the data & movie
global I;
global O;

[dataFlag movieFlag t_step_4_graph] = deal(I.dataFlag, I.movieFlag, I.t_step_4_graph);

if (dataFlag == true)
    % calls out variables from global O & I
    [svec Equilibrium eigenvalues_14x14 largest_eigenvalue_14x14 ...
        oscillation_freq_14x14] = deal(O.svec, O.Equilibrium, ...
        O.eigenvalues_14x14, O.largest_eigenvalue_14x14, O.oscillation_freq_14x14);
    [tend Nspace deltat deltax lcortex t_step_4_tplot] = deal(...
        I.tend, I.Nspace, I.deltat, I.deltax, I.lcortex, I.t_step_4_tplot);
    % saves the data with stability test to file "firing_rates_etc"
    save(I.DataFileName, ...
        'svec', 'tend', 'Nspace', 'deltat', 'deltax', 'lcortex', ...
        't_step_4_tplot', 'Equilibrium', 'eigenvalues_14x14', ...
        'largest_eigenvalue_14x14', 'oscillation_freq_14x14');
end

if (t_step_4_graph ~= 0) && (movieFlag == true) % Saves the movie
    Mmovie = O.Mmovie; % calls out the frames from global
    save(I.MovieFileName, 'Mmovie');
    movie2avi(Mmovie,I.MovieFileName);
end

return
```

C.12 Power_Spectrum.m

```
function Power_Specturm
%
% Jay Chang 4 October 2006. Code for calculating power spectrum
%
% Marcus Wilson comments 5 october 2006. Still need to put into epochs
% (discard first?)
% Calculate Nyqvist frequency and use this to cut-off xaxis.

clear all;

testing_mode = 0;   % 1, 2 or 3 are testing mode, 0 is normal mode.

%************************************************************************
% The section below is the testing cases for power spectrum
```

```
if testing_mode ~= 0
    warning('testing mode activated!!! (function Power_Specturm)');
    fprintf('\n press any key to proceed testing or Ctrl-C to stop the program \n\n\n');
    pause
    % uses these values for the parameters below for the testing cases.
    Nspace = 32;
    deltat = 0.2*1e-3;
    lcortex = 0.5*1e3;
    deltax = lcortex/Nspace;
    t_step_4_tplot = 10;
    tend = 10000;

    t_increment = deltat * t_step_4_tplot;
    T = tend*deltat;

    testing_time = 0:t_increment:T;
    settling_time = 0.2;
    division_period = 0.5;

switch testing_mode

    case 1
        % This case gives a sine wave signal with constant frequency.
        testing_frequency = 20; % unit in Hz
        xy = 10*sin(2*pi*testing_frequency*testing_time);
        for i = 1:length(testing_time)
            svec(1:Nspace,1:Nspace,1,i) = xy(i);
        end

    case 2
        % This case gives a signal with frequency proportional to the time

        % frequency ranges from 1 - 10 Hz
        testing_frequency = linspace(1,10,length(testing_time));

        % frequency is proportional to the time in this case
        y = sin(2*pi*testing_frequency.*testing_time);
        for i = 1:length(testing_time)
            svec(:,:,1,i) = peaks(32) * y(i) -65;
        end

    case 3
        % This case gives a signal with frequency proprtional to the time, and also
        % has ripple in space.

        % frequency ranges from 1 - 10 Hz
        testing_frequency = linspace(1,10,length(testing_time));

        % frequency is proportional to the time in this case
        y = sin(2*pi*testing_frequency.*testing_time);
```

```
        for i = 1:length(testing_time)
            if (i < 200) && (i >= 0)
                testing_space = zeros(32,32);
                testing_space(1:5,1:5) = 10;
            elseif (i < 400) && (i >= 200)
                testing_space = zeros(32,32);
                testing_space(6:10,6:10) = 10;
            elseif (i < 600) && (i >= 400)
                testing_space = zeros(32,32);
                testing_space(11:15,11:15) = 10;
            elseif (i < 800) && (i >= 600)
                testing_space = zeros(32,32);
                testing_space(16:20,16:20) = 10;
            else
                testing_space = zeros(32,32);
                testing_space(21:25,21:25) = 10;
            end
            svec(:,:,1,i) = testing_space * y(i) -65;
        end

    otherwise
        error('Unknown Power Spectrum testing case!');
end
%*************************************************************************
% read in the data simulated by function Cortex_System
else
        global I;
        I = Input;              % Set the global parameters for inputs
        load(I.DataFileName);               % load data from Cortex_System

        % the data within the settling time will be ignored for processing
        % (unit in second)
        settling_time = I.settling_time;

        % user-defined value for fourier transform time interval (unit in
        % second)
        division_period = I.division_period;
        Variable = I.Analyzed_Variable;
end

% time between each RECORDED datapoint
t_increment = deltat * t_step_4_tplot;

% *************************************************************************
% the codes here read-in the data after the settling time(wait for signal
% to settle)

% convert settling_time from seconds to No. of time elements in the vector
settling_time = settling_time / t_increment;
if settling_time > length(svec(1,1,1,:))
```

```
    error('settling time is longer than the data time length, please inputs a smaller value for settling time
else
    V(:,:,:) = svec(:,:,Variable,(settling_time:end));% read in the variable
end
% ************************************************************************

% convert division time interval in seconds to No. of elements
division_size = division_period / t_increment;

% the total No. of time elements in the data
total_time_steps = length(V(1,1,:));

% ************************************************************************
% the if statement below is for preventing error occur if user inputs a
% division period larger than the total time of the data.
if division_size > total_time_steps
    division_size = total_time_steps;
    % if user input a division period larger than the total time of the
    % data, then the division period will be changed to be the same length
    % as the total time of the data.
end
% ************************************************************************

total_divisions = floor(total_time_steps / division_size);
% only do the fourier transform for the division with full period time
% length. (ignore the numbers of data in the end if they are not enough to
% complete a full division.)

nyquist_time = 1/(2*t_increment); % Nyquist angular frequency for time
nyquist_space = 2*pi/(2*deltax); % Nyquist wavevector for space

for division = 1:total_divisions

V_division(:,:,:) = ...
    V(:,:,((((division-1)*division_size)+1):(division*division_size)));
% read-in the data for the division to do the fourier transform.

% fourier transform of temporal component
i = 1;
for x = 1:Nspace
    for y = 1:Nspace
        Vt(1,:) = V_division(x,y,:);           % Ve discrete
        Vk = fft(Vt);                           % fourier transform
        Vk_square = Vk .* conj(Vk);             % Power, |Vk|^2
        scale_factor = (t_increment/(2*pi))*sum(Vt.^2) / sum(Vk_square);
        P_temporal(i,:) = Vk_square * scale_factor; % corrected Power
        i = i+1;
    end
end
```

```
Power_temporal(division,:) = mean(P_temporal); % average Power

%remove constant term so we can see the rest
Power_temporal(division,1) = NaN;

frequency(1:length(Power_temporal(division,:))) = ...
    ((1:length(Power_temporal(division,:)))-1)/division_period;
% calculates the frequency vector. (divide by the division period time)

figure(1); clf;
subplot(121)
plot(frequency,Power_temporal(division,:), 'linewidth', 2)
xlabel('Frequency (Hz)', 'fontsize', 16)
ylabel('Power fluctuation, |V_k|^2  (mV^2 / (Hz) ) ', 'fontsize', 16)
title(['Power Spectrum, division = ', num2str(division)], 'fontsize', 18)
set(gca,'xlim',[0 nyquist_time])
set(gca,'xtick',[0 0.2 0.4 0.6 0.8 1]*nyquist_time)  % more meaningful axis

%************************************************************************
% Now we do the fourier transform of spatial components
i = 1;
% The for loop here does fourier transform through x-axis
for x = 1:Nspace
    V_ij(:,:) = V_division(x,:,:);         % V_ij here = Ve(y,t)
    V_qxqy = fft2(V_ij);                    % do the 2-D fourier transform
    V_qxqy_square = V_qxqy .* conj(V_qxqy); %Power
    scale_factor = (deltax*t_increment / (4*pi^2) )*sum(sum(V_ij.^2)) / ...
        sum(sum(V_qxqy_square));
    P_spatial(i,:,:) = V_qxqy_square * scale_factor;
    i = i+1;
end

% The for loop here does fourier transform through y-axis
for y = 1:Nspace
    V_ij(:,:) = V_division(:,y,:);         % V_ij here = Ve(x,t)
    V_qxqy = fft2(V_ij);                    % do the 2-D fourier transform
    V_qxqy_square = V_qxqy .* conj(V_qxqy); %Power
    scale_factor = (deltax*t_increment / (4*pi^2) )*sum(sum(V_ij.^2)) / ...
        sum(sum(V_qxqy_square));
    P_spatial(i,:,:) = V_qxqy_square * scale_factor;
    i = i+1;
end

Power_spatial(:,:) = mean(P_spatial);       % average Power
Power_spatial(1,:) = NaN;  %remove constant term so we can see the rest

% calculates the spatial frequency
q(1:Nspace) = 2*pi*((1:Nspace)-1)/lcortex;

% creates q matrix for scaling power
```

```
q_matrix = q'*ones(1,length(Power_spatial(1,:)));

Power_spatial = Power_spatial ./ q_matrix; % scaling power

% saves power data for averaging later on
Power_spatial_2(division,:,:) = Power_spatial(:,:);

subplot(122)
mesh(frequency,q,Power_spatial)
xlabel('Frequency (Hz)', 'fontsize', 16)
ylabel('q (mm^{-1})', 'fontsize', 16)
zlabel('Power fluctuation (mV^2 / (Hz) / (mm^{-1})', 'fontsize', 16)
title(['Power Spectrum, division = ', num2str(division)], 'fontsize', 18)
set(gca,'xlim',[0 nyquist_time])
set(gca,'ylim', [0 nyquist_space])

pause
end

% plot out the power spectrum if there are more than one division.
if division > 1
    Power_temporal_2 = mean(Power_temporal);
division = (1:total_divisions); % gets the vector for division
Power_temporal = Power_temporal';
figure(1); clf; colormap(jet)
%subplot(221)
%mesh(division,frequency,Power_temporal)
surf(division,frequency,Power_temporal)
set(gca,'xlim',[division(1) division(end)]);
set(gca,'ylim',[frequency(2) nyquist_time]);
xlabel('Division', 'fontsize', 16)
ylabel('Frequency (Hz)', 'fontsize', 16)
title('Power Spectrum', 'fontsize', 18)
view(0,90);

%subplot(222)
%plot(frequency,Power_temporal_2, 'linewidth', 2)
%xlabel('Frequency (Hz)', 'fontsize', 16)
%ylabel('Power fluctuation, |V_k|^2  (mV^2 / (Hz) ) ', 'fontsize', 16)
%title('Average Power Spectrum', 'fontsize', 18)
%set(gca,'xlim',[0 nyquist_time])
%set(gca,'xtick',[0 0.2 0.4 0.6 0.8 1]*nyquist_time)

%Power_spatial_3(:,:) = mean(Power_spatial_2);
%subplot(223)
%mesh(frequency,q,Power_spatial_3)
%xlabel('Frequency (Hz)', 'fontsize', 16)
%ylabel('q (mm^{-1})', 'fontsize', 16)
%zlabel('Power fluctuation (mV^2 / (Hz) / (mm^{-1})', 'fontsize', 16)
%title('Average Power Spectrum', 'fontsize', 18)
```

```
%set(gca,'xlim',[0 nyquist_time])
%set(gca,'ylim', [0 nyquist_space])
end

return
```

C.13 Variable_Variation.m

```
function Variable_Variation
clear all;

global I;
I = Input;              % Set the global parameters for inputs

load(I.DataFileName);      % load data from Cortex_System
t_increment = deltat * t_step_4_tplot;%time between each RECORDED datapoint

[user_defined_point_x user_defined_point_y Analyzed_Variable] = deal(...
    I.user_defined_point_x, I.user_defined_point_y, I.Analyzed_Variable);

% the data within the settling time will be ignored for processing (unit in
% second)
settling_time = I.settling_time;

% convert settling_time from seconds to No. of time elements in the vector
settling_time = settling_time / t_increment;
if settling_time > length(svec(1,1,1,:))
    error('settling time is longer than the data time length, please inputs a smaller value for settli
else
    % read in Ve with user-defined y-axis
    Variable_y(:,:) = svec(:, user_defined_point_y, Analyzed_Variable, ...
        (settling_time:end));

    % read in Ve with a user-defined point
    Variable_xy(1,:) = svec(user_defined_point_x, user_defined_point_y, ...
        Analyzed_Variable, (settling_time:end));
end

space = (1:Nspace)*deltax/1000; % calculates the space vector
time = (1:length(Variable_y(1,:)))*t_increment;% calculates the time vector


figure(1); clf;

subplot(121)
plot(time,Variable_xy)
set(gca,'xlim',[0 time(end)]);
xlabel('time (s)', 'fontsize', 16)
ylabel('Ve(t) (mV)', 'fontsize', 16)
title('excitatory soma voltage vs. time', 'fontsize', 18)
```

```
subplot(122)
mesh(time,space,Variable_y)
set(gca,'xlim',[0 time(end)]);
set(gca,'ylim',[space(1) space(end)]);
xlabel('time (s)', 'fontsize', 16)
ylabel('space(y-axis) (m)', 'fontsize', 16)
title('excitatory soma voltage vs. time & space(y-axis)', 'fontsize', 18)
colormap(gray);
caxis([-65 -50]);
view(0,90);
return
```

References

[1] Internet WWW page at *http://life.nthu.edu.tw/~g864264/Neuroscience/neuron/intro.htm* last accessed 15/03/07.

[2] M.T. Wilson, Moira L. Steyn-Ross, D.A. Steyn-Ross, and J.W. Sleigh, *Predictions and simulations of cortical dynamics during natural sleep using a continuum approach*, Physical Review E, vol.72, pp 051910-1 to 051910-14 (2005).

[3] M.T. Wilson, Moira L. Steyn-Ross, D.A. Steyn-Ross, and J.W. Sleigh, *Predictions and simulations of cortical dynamics during natural sleep using a continuum approach. Supplementary material: calculation of the power spectra*, Physical Review E, vol.72, pp 051910-1 to 051910-14 (2005).

[4] Internet WWW page at *http://webspace.ship.edu/cgboer/theneuron.html* last accessed 21/03/07.

[5] Internet WWW page at *http://www.ncu.edu.tw/~ncu5200/en/f_01.php* last accessed 27/03/07.

[6] Adam Cone (2002), *Towards Modeling Neuron Population Responses in the Visual Cortex: A Comparison of the Monte-Carlo and Population-Density Methods*. Summer Research Project, New York University.

[7] Daniel Tranchina, Felix Apfaltrer and Cheng Ly, *Population density methods for stochastic neurons with realistic synaptic kinetics: Firing rate dynamics and fast computational methods*, Network: Computation in Neural Systems, 17:4, 373 - 418

[8] Cheng Ly, *Critical Analysis of Dimension Reduction for a Moment Closure Method in a Population Density Method*, http://www.mbi.osu.edu/postdocworkshop/wyrmbabstracts.html

[9] W. J. Freeman, in *Induced Rhythms of the Brain*, edited by E. Basar and T. H. Bullock (Birkhaeuser, Boston, 1992), pp. 183-199

[10] P. L. Nunez, Math. Biosci. 21, 279 (1974)

[11] J. J. Wright and D. T. J. Liley, Behav. Brain Sci. 19, 285 (1996).

[12] P. A. Robinson, C. J. Rennie, and J. J. Wright, Phys. Rev. E56, 826 (1997).

[13] D. T. J. Liley, P. J. Cadusch, and J. J. Wright, *Neurocomputing* 26-27, 795(1999).

[14] C. J. Rennie, J. J. Wright, and P. A. Robinson, J. Theor. Biol. 205, 17 (2000).

[15] D. A. Steyn-Ross, M. L. Steyn-Ross, J. W. Sleigh, M. T. Wilson, I. P. Gillies, and J. J. Wright, *J. of Biophysics*, Journal of Biophysics (31) 543-565 (2005).

[16] D. A. Steyn-Ross, (2002) *Modelling the Anaestheto-Dynamic Phase Transition of the Cerebral Cortex*, PhD Thesis, University of Waikaton, New Zealand.

[17] Ian Sommerville, *Software engineering*, (2004).

[18] P. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations*, Sprinter, Berlin, New York (1992).

[19] M. E. Hasselmo, *Behavioral Brain Research*, E 67, 1 (1995).

[20] Steyn-Ross DA, Steyn-Ross ML, Sleigh JW, Wilson MT, Gillies IP and Wright JJ, *The sleep-cycle modelled as a cortical phase transition*, Journal of Biological Physics 31:567 569 (2005)

[21] Banks MI, Pearce RA, *Dual actions of volatile anesthetics on GABA(A) IPSCs: dissociation of blocking and prolonging effects*, Anesthesiology 1999; 90: 120-34

[22] F Amzica and M. Steriade, *The functional significance of the k-complex*, Sleep Med. Rev. 6 139-149 (2002)