Mathematical Modelling in Neurophysiology: Neuronal Geometry in the Construction of Neuronal Models

by

Gayle Tucker

A thesis submitted to the
Faculty of Biomedical and Life Sciences
at the University of Glasgow
for the degree of
Doctor of Philosophy

December 2004

Summary

The underlying theme of this thesis is that neuronal morphology influences neuronal behaviour. Three distinct but related projects in the application of mathematical models to neurophysiology are presented. The first problem is an investigation into the source of the discrepancy between the observed conduction speed of the propagated action potential in the squid giant axon, and its value predicted on the basis of the Hodgkin-Huxley membrane model. It is shown that measurement error and biological variability cannot explain the discrepancy, nor can the use of a three-dimensional model to represent the squid giant axon. If the propagated action potential achieved the travelling wave speed in the experimental apparatus, as assumed implicitly by Hodgkin and Huxley, then it is suggested that the model of the membrane kinetics requires modification. The second problem involves the generalisation of Rall's equivalent cylinder to the equivalent cable. The equivalent cable is an unbranched structure with electrotonic length equal to the sum of the electrotonic lengths of the segments of the original branched structure, and an associated bijective mapping relating currents on the original branched structure to those on the cable. The equivalent cable is derived analytically and can be applied to any branched dendrite, unlike the Rall equivalent cylinder, which only exists for dendrites satisfying very restrictive morphological constraints. Furthermore, the bijective mapping generated in the construction of the equivalent cable can be used to investigate the role of dendritic morphology in shaping neuronal behaviour. Examples of equivalent cables are given for spinal interneurons from the dorsal horn of the spinal cord. The third problem develops a new procedure to simulate neuronal morphology from a sample of neurons of the same type. It is conjectured that neurons may be simulated on the basis of the single assumption that they are composed of uniform dendritic sections with joint distribution of diameter and length that is independent of location in a dendritic tree. This assumption, in combination with the kernel density estimation technique, is used to construct samples of simulated interneurons from samples of real interneurons, and the procedure is successful in predicting features of the original samples that are not assumed by the construction process.

Acknowledgements

My degree was funded by the Future and Emerging Technologies programme (IST-FET) of the European Community, under Grant IST-2000-28027 (POETIC). The information provided is the sole responsibility of the author and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication.

In addition, I would like to thank the following people who have contributed, in their own way, to my thesis:

My two supervisors: Professor Jay Rosenberg and Dr Kenneth Lindsay. Between them, they knew the answer to almost every question I could think of and showed great patience while waiting for the penny to drop inside my head. Although I think the latter may have been aided by freshly ground coffee, and occasionally some cake.

Dr Jim Morrison for providing moments of light relief, small quantities of milk and who I thank for telling me about this opportunity in the first place.

My family and friends, who quite openly had no idea of what I was doing but supported me nonetheless.

And Martin for being there always.

Contents

1 Introduction 7

2 Action Potentials 11

2.1 Introduction 11

2.2 Mathematical and physical preliminaries 14

2.2.1 Definitions 14

2.3 Derivation of the neuronal field equations 15

2.3.1 Dispersion of charge 17

2.3.2 Diffusion of magnetic field 18

2.3.3 Maxwell's equations in neuronal modelling 18

2.3.4 Equations for a three-dimensional axonal cylinder 19

2.3.5 Identification of the one-dimensional membrane potential 20

2.3.6 Three-dimensional axon with Hodgkin-Huxley kinetics 23

2.4 Measurement error and biological variability 26

2.4.1 Measurement error 26

2.4.2 Biological variability 28

2.4.3 Results of simulation exercises 29

2.5 The travelling wave assumption 33

2.5.1 A possible explanation for the discrepancy between the observed and predicted conduction speed 35

2.6 Trains of action potentials 36
2.6.1 The dispersive relationship 36
2.6.2 Activation variables 39
2.7 Conclusions 42

3 Neuron data and terminology 43

4 Equivalent cable 46
4.1 Introduction 46
4.2 Constructing the model dendrite 49
4.2.1 Mathematical model of a uniform cable 51
4.2.2 Model equations for a branched dendrite 53
4.3 Notation 55
4.4 Analytical development of the equivalent cable 56
4.4.1 Construction of the discrete model dendrite 56
4.4.2 The piecewise uniform cable 57
4.4.3 Symmetrising a cable matrix 59
4.4.4 Structure of tree matrices 61
4.4.5 Symmetrising the tree matrix 64
4.4.6 Tri-diagonalising the symmetric tree matrix 64
4.4.7 Mapping of potentials and currents from tree to equivalent cable . . 65
4.4.8 Construction of the equivalent cable 66
4.4.9 Summary - Concept of the equivalent cable 68
4.5 Analytical construction of an equivalent cable 69
4.5.1 A simple Rall branch point 69
4.5.2 An asymmetric Y-junction 74
4.5.3 A symmetric Y-junction 78
4.6 Application of the equivalent cable to spinal interneurons 87

4.6.1 Distribution of contacts 87
4.7 Conclusions 92
4.8 Mathematical appendix 93
4.8.1 Householder procedure 93
4.8.2 Estimating the density of contacts 96

5 Building the typical neuron **98**

5.1 Introduction 98
5.1.1 Some recent models of dendritic morphology 100
5.1.2 A new approach to the simulation of dendritic morphology 102
5.2 Mathematical preliminaries 103
5.2.1 Probability density function 103
5.2.2 Joint probability density function 103
5.3 A procedure for simulating a sample of dendrites 105
5.4 Estimating distributions 108
5.4.1 Multivariate kernel estimates of density 110
5.4.2 Marginal and conditional densities 111
5.4.3 Bandwidth selection 112
5.4.4 Comparison of distributions 113
5.4.5 Drawing from a random distribution 115
5.5 Validation of the model assumption 117
5.6 Formal procedure for constructing a dendrite 119
5.7 Results 123
5.7.1 First test of the construction procedure 124
5.7.2 Comparison of observed and simulated probability densities 125
5.7.3 Summary 132
5.7.4 A further development 133

5.8 Conclusions 134

6 Concluding remarks and future work **135**

References **137**

A Associated programs **143**

A.1 Action potentials 143

A.1.1 HHPotVel.c 143

A.1.2 HHDisperse.c 152

A.2 Extract data from Neurolucida file 159

A.2.1 BuildNeuron.c 159

A.3 Equivalent cables 173

A.3.1 MapContacts.c 173

A.3.2 CumulativeCurrent.c 182

A.3.3 Lagrange.c 184

A.4 Building the typical neuron 188

A.4.1 MyeTypicalNeuron.c 188

A.4.2 BiVarKernels.c 203

A.4.3 SimStats.c 206

A.5 Hodgkin-Huxley rate functions 208

A.6 Utility programs 211

A.7 Bandwidth calculations 220

A.8 Fast Fourier Transform 222

A.9 Differential equation solver 229

Chapter 1

Introduction

> *A good model is one that succeeds to reduce the complexity of the modelled system significantly while still preserving its essential features.* (Segev, 1992)

The underlying theme of this thesis is that neuronal morphology influences neuronal behaviour. This thesis presents three distinct but related projects in the application of mathematical models to neurophysiology. The problems to be discussed are:-

(a) an investigation into the source of the discrepancy between the observed conduction speed of the propagated action potential in the squid giant axon, and its value predicted on the basis of the Hodgkin-Huxley membrane model;

(b) the generalisation of Rall's equivalent cylinder to arbitrary branched dendrites;

(c) the development of a new procedure to simulate neuronal morphology from a sample of neurons of the same type.

In each of these examples of mathematical modelling in neurophysiology, the original work made a significant contribution to neurophysiology. For example, the Hodgkin-Huxley model (1952d) and the supporting experimental work (Hodgkin, Huxley and Katz, 1952; Hodgkin and Huxley, 1952a,b,c) provided the benchmark combination of theoretical and experimental practice against which future work can be measured. Nevertheless in each of the examples (a)-(c) there remain unresolved issues, some of which could not be resolved at the time of the original work due either to limitations in the computational tools or the requirement of conceptual advances. The Hodgkin-Huxley model is an example of the

former, and the Rall equivalent cylinder is an example of the latter. In view of the diverse subject matter of this thesis, this introduction will provide a preliminary background to the content of each project, with the introduction of each chapter providing a more comprehensive description of the individual projects.

Hodgkin and Huxley carried out a series of experiments, some in collaboration with Bernard Katz, which culminated in a model for the generation of an action potential in the squid giant axon (Hodgkin *et al.*, 1952; Hodgkin and Huxley 1952a,b,c,d). Hodgkin and Huxley were awarded the Nobel prize for this work in 1963. Of the experimental observations predicted by the model, the prediction of the conduction speed of the propagated action potential is the strongest test of the model, since this speed is independent of the conditions under which the membrane model was derived. In this test, there is an 11-12% discrepancy between the observed and predicted conduction speeds, and although this discrepancy was acceptable at the time, its source remains to be explained. This result has been unquestioned since it was published in 1952. Cole (1968) refers to the discrepancy suggesting that it may have been the result of assumptions and approximations in the experimental process and is therefore an acceptable result under the circumstances. However, given the tight experimental conditions imposed by Hodgkin and Huxley, whereby all errors were known and small, this explanation seems unlikely. Chapter 2 examines Cole's explanation and other possible explanations for the discrepancy between the observed conduction speed of the propagated action potential in the squid giant axon and the conduction speed predicted by the Hodgkin-Huxley membrane model.

The predominant view of the role of dendritic morphology in the mid-twentieth century, based on intracellular recordings from motor neurons, was that dendritic structure was unimportant in conditioning the behaviour of the motor neuron. Jack and Redman (1995) outline the controversy between the dominant view held by Eccles, that is, that dendritic form is not important, and the position taken by Rall, that dendritic morphology was important and must be taken into account to understand the behaviour of neurons. Once Rall established this view of the dendrite, the investigation of the role of dendritic morphology in shaping neuronal behaviour became an important issue. Given the complex and varied morphology of dendritic trees, building a model of their structure is not straightforward. Using linear cable theory to describe each cylindrical segment of a dendritic tree in combination with several restrictive conditions, Rall derived the equivalent cylinder. The equivalent cylinder produces the same response at the soma as that of the branched tree

when current is injected at the same electrotonic distance from the soma. The conditions imposed by Rall for this equivalence to be valid are often unrealistic when applied to real neurons. Particularly unrealistic were the requirements that the electrotonic distance from any branch point to a dendritic tip connected to that branch point be identical, and that dendritic sections meeting at a branch point should satisfy the 3/2 power law, by which is meant that the sum of the daughter diameters raised to the 3/2 power must equal the parent diameter raised to the 3/2 power. The equivalent cylinder generated under these assumptions is not truly equivalent to the original dendrite from which it is constructed, because input to the equivalent cylinder cannot be uniquely associated with input on the original dendritic tree. Chapter 4 generalises the Rall equivalent cylinder to the equivalent cable by relaxing all the constraints imposed by Rall. The procedure is analytical and leads to a bijective mapping between input on the original tree and that on the equivalent cable. Examples of equivalent cables are given for spinal interneurons from the dorsal horn of the spinal cord that receive myelinated and unmyelinated afferent input.

The first accurate description of neuronal anatomy was provided by Cajal using a light microscope to examine Golgi stained neurons (see Cajal, 1952). From these studies, he accurately theorised that the neuron was the basic unit of the nervous system, and furthermore, that neurons communicated by the spread of a signal from the dendritic tree to the soma and onto the axon from where the signal is transmitted to the next neuron. It was clear from Cajal's work that neurons formed diverse morphological structures ranging from the densely branched Purkinje cell to the sparsely branched spinal interneurons considered in this thesis. Various studies have focused on the identification of a canonical set of properties that characterise neuronal morphology. Hillman (1979) was at the forefront of this research when he described seven fundamental parameters for the size and shape of dendritic trees, and latterly Tamori (1993) introduced two additional parameters to characterise the three-dimensional orientation of neurons. Burke, Marks and Ulfhake (1992) and Ascoli and Kirchmar (2000) simulated neuron morphology using some of the parameters identified by Hillman (1979) and Tamori (1993) with varying degrees of success. For example, Burke *et al.* (1992) find that they need to introduce additional correction factors to improve the agreement between their simulated and real neurons. Chapter 5 sets out a novel procedure based on the single assumption that neurons are composed of uniform dendritic sections with joint distribution of diameter and length that is independent of location in a dendritic tree. This assumption, in combination with the kernel density esti-

mation technique, is used to construct samples of simulated neurons from a sample of real neurons.

Chapter 2

A comparison of the three-dimensional and one-dimensional treatment of action potentials

2.1 Introduction

The Hodgkin-Huxley model for membrane kinetics in the squid giant axon has become the standard model in neurophysiology despite an 11-12% discrepancy between theory and observation in a critical test of the model. Specifically, the prediction of the conduction velocity of the propagated action potential. The intention of the chapter is to arrive at an explanation for this discrepancy by considering the standard assumption of a one-dimensional model and the implications of applying a three-dimensional model to the same problem. Other possible explanations for the discrepancy will also be examined. The development of the three-dimensional model will allow a further investigation into the relationship between the speed of the action potential and its wavelength.

The development of the Hodgkin-Huxley membrane model marked a turning point in neurophysiology. The precise and systematic quantitative analysis of the ionic currents of the squid giant axon led to an accurate and robust model that has stood the test of time since its publication in 1952. Hodgkin and Huxley (1952d) derived a set of partial differential

equations that can be used to describe the evolution of the membrane potential in the squid giant axon. By assuming a travelling wave solution for the set of partial differential equations, namely an action potential with fixed profile moving at a constant speed, the equations are simplified to a set of ordinary differential equations and subsequently solved numerically by hand calculation. This procedure contains obvious sources of error that may explain the discrepancy; possibilities are transcription errors in copying the results of calculations from the hand calculator, or rounding error introduced by the limited accuracy of the hand calculator. With the development of more accurate digital computers, Cooley and Dodge (1966) and Huxley (1959) himself have retested the model and found the original calculations to be accurate. Therefore we can exclude immediately numerical inaccuracy as a possible source of the 11-12% discrepancy.

Once numerical inaccuracies have been ruled out, one must consider the explicit and implicit assumptions made in the construction of the model. The most significant implicit assumption made is that of a one-dimensional axon. Hodgkin and Huxley used the one-dimensional cable equation to describe the behaviour of the three-dimensional axon, a feature of the model that has never been questioned. In fairness, the body of work published prior to Hodgkin and Huxley, for example Hodgkin and Rushton (1946), Davis and Lorente de No (1947), Hermann (1884) and even Lord Kelvin's (1855) original work on cable theory all implicitly assume one-dimensional structures.

However, subsequent work also supports the assumption of a one-dimensional axon. The approach commonly taken is a one-dimensional approximation of a three-dimensional structure, rather than deriving the one-dimensional object from the three-dimensional model. Rall (1959) assumes that the membrane potential is a solution of Laplaces equation while Jack, Noble and Tsien (1975) apply a geometrical argument to investigate the incorporation of taper into a one-dimensional model. This leaves the model vulnerable to difficulties, for example, a change in the surface area per unit length will be constant if the axon is a cylinder, however, if the axon tapers the change in surface area is no longer proportional to length. As will be shown in the development of the three-dimensional model, there are features found in the simplification of the three-dimensional model that are neither geometric nor explicit and therefore it is not obvious how they might be included in the generalised one-dimensional model. These additional features describe radial currents in the axon, and it is their existence that compromises the validity of one-dimensional models.

Sixteen years after the publication of the Hodgkin-Huxley membrane model, Cole (1968) noted that the assumption of the absence of radial gradients in Kelvin's work may be invalid in cables although he did not question this assumption in the case of the squid axon. It is interesting to note that Cole recognised that this assumption may not be valid for axons with large diameters, and at the same time commented on the discrepancy between the predicted and observed speeds of the propagated action potential in the Hodgkin-Huxley membrane model, while not recognising that the two phenomena might be related. It is clear therefore that both the predominant view at the time of Hodgkin and Huxley's work and subsequent work by Rall (1962) reached the same conclusion; namely, that it is safe to ignore the effects of radial gradients in axonal and dendritic models.

An explicit assumption that requires verification is the validity of the travelling wave assumption. For the travelling wave assumption to be valid, the speed of the action potential must reach the travelling wave speed over the length of axon over which it is measured. In addition, one must also consider measurement error and biological variability as possible sources of error. Due to the diligent reporting of Hodgkin and Huxley, the error bounds for all of the measured parameters in their experiments are known. It is possible to manipulate both measurement error and biological variability through simulation exercises, taking into account all possible combinations to assess the effect on the conduction speed.

An adequate test of the one-dimensional assumption takes Maxwell's equations as the starting point of the model. The particularisation of these equations to the case of a cylindrical axon of known diameter forms an important part of this chapter. It is shown that Maxwell's equations when particularised to the material properties of an axon and its geometry lead to Laplace's equation. Maxwell's equations describe the fundamental features of the three-dimensional model from which a one-dimensional model of the axon incorporating all its biophysical and geometrical properties can be extracted. The reduction of the three-dimensional model to a one-dimensional representation reveals irreducible features of the three-dimensional model that are a consequence of radial gradients and not geometrical features. Using the one-dimensional representation derived from the three-dimensional model the conduction speed of the propagating action potential can be calculated without the travelling wave assumption. By solving Laplace's equation for the potential distributions in the interior and exterior regions, an investigation can also be carried out on the dispersive relationship for action potentials and their implications for signalling in axons.

2.2 Mathematical and physical preliminaries

This section lays out the mathematical methods and motivations used to construct the three-dimensional model of an axon and solve the associated equations.

2.2.1 Definitions

Before the particularisation of Maxwell's equations, it is necessary to lay out some definitions and identities from vector calculus. Let $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ form a right-handed triad of mutually orthogonal unit vectors. By this we mean that

$$\begin{gathered}\mathbf{i}\,.\,\mathbf{i} = \mathbf{j}\,.\,\mathbf{j} = \mathbf{k}\,.\,\mathbf{k} = 1 \\ \mathbf{i}\times\mathbf{j} = \mathbf{k} \qquad \mathbf{j}\times\mathbf{k} = \mathbf{i} \qquad \mathbf{k}\times\mathbf{i} = \mathbf{j}\end{gathered} \tag{2.1}$$

where '.' and '$\times$' denote the scalar and cross-products of vectors in $\mathbb{R}^3$. Let $f = f(x, y, z)$ be a scalar function of position then the gradient of f, denoted $\operatorname{grad} f$, is the vector

$$\operatorname{grad} f = \frac{\partial f}{\partial x}\mathbf{i} + \frac{\partial f}{\partial y}\mathbf{j} + \frac{\partial f}{\partial z}\mathbf{k}.$$

The gradient of f in the direction $\mathbf{n}$, where $\mathbf{n}$ is a unit vector, is given by

$$\frac{\partial f}{\partial n} = \mathbf{n}\,.\operatorname{grad} f.$$

In particular, if $\mathbf{n}$ is the normal to a surface S, then this gradient is called the normal derivative of f at (x, y, z) on the surface S.

The divergence of a vector field $\mathbf{F} = F_1\,\mathbf{i} + F_2\,\mathbf{j} + F_3\,\mathbf{k}$ is defined by

$$\operatorname{div}\mathbf{F} = \frac{\partial F_1}{\partial x} + \frac{\partial F_2}{\partial y} + \frac{\partial F_3}{\partial z}.$$

Given any volume of $\mathcal{V}$ with surface $\partial\mathcal{V}$, Gauss's theorem asserts that

$$\int_{\mathcal{V}} \operatorname{div}\mathbf{F}\, dV = \int_{\partial\mathcal{V}} \mathbf{F}\,.\,\mathbf{n}\, dS,$$

so that $\operatorname{div}\mathbf{F}$ may be interpreted as the distribution of sources of $\mathbf{F}$ within the volume $\mathcal{V}$ that will give rise to flux $\mathbf{F}$ on the surface of the volume.

The curl of a vector field $\mathbf{F} = F_1\,\mathbf{i} + F_2\,\mathbf{j} + F_3\,\mathbf{k}$ is defined by

$$\operatorname{curl}\mathbf{F} = \left(\frac{\partial F_3}{\partial y} - \frac{\partial F_2}{\partial z}\right)\mathbf{i} + \left(\frac{\partial F_1}{\partial z} - \frac{\partial F_3}{\partial x}\right)\mathbf{j} + \left(\frac{\partial F_2}{\partial x} - \frac{\partial F_1}{\partial y}\right)\mathbf{k}. \tag{2.2}$$

Given any closed curve C in space, Stoke's theorem asserts that

$$\int_C \mathbf{F} \,.\, \mathrm{dl} = \int_S \mathbf{n} \,.\, \mathrm{curl}\mathbf{F}\, dS$$

for all surfaces S for which C is a boundary curve. Thus curl $\mathbf{F}$ is a measure of the rotation of the vector field $\mathbf{F}$. For example, if $\mathbf{E}$ is an electric field, then curl $\mathbf{E}$ would measure the extent to which the field is not conservative, and if curl $\mathbf{E} = \mathbf{0}$ then $\mathbf{E}$ is derivable as the gradient of a potential function.

The operators grad, div and curl satisfy the important identities

$$\begin{aligned} &\mathrm{div}\ \mathrm{curl}\, \mathbf{F} = 0 \\ &\mathrm{curl}\ \mathrm{curl}\, \mathbf{F} = \mathrm{grad}\ \mathrm{div}\, \mathbf{F} - \triangle \mathbf{F} \\ &\mathrm{curl}\ \mathrm{grad}\, f = \mathbf{0} \end{aligned}$$

where $\triangle \mathbf{F}$ is called the Laplacian of $\mathbf{F}$. In particular, if ϕ is expressed as a scalar function of the Cartesian coordinates (x, y, z) then

$$\triangle \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2},$$

where $\triangle \phi = 0$ is called Laplace's equation.

2.3 Derivation of the neuronal field equations

The starting point for the three-dimensional model of a dendrite in a stationary medium is Maxwell's equations

$$\mathrm{div}\ \mathbf{D} = \rho, \tag{2.3}$$

$$\mathrm{div}\ \mathbf{B} = 0, \tag{2.4}$$

$$\mathrm{curl}\ \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}, \tag{2.5}$$

$$\mathrm{curl}\ \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}. \tag{2.6}$$

In these equations, $\mathbf{E}$ (mV/mm) is the electric field strength, $\mathbf{D} = \epsilon_0 \mathbf{E} + \mathbf{P}$ ($\mu C/\mathrm{mm}^2$) is the electric displacement field with $\mathbf{P}$ ($\mu C/\mathrm{mm}^2$) the polarisation of the medium and $\epsilon_0 = (36\pi)^{-1}\ \mu\mathrm{F/mm}$ is the permittivity of free space. The scalar function ρ (nC/mm^3) is the density of free charge in the medium, $\mathbf{J}$ (μA/mm^2) is the current density, $\mathbf{H}$ (mA/mm) is the magnetic field strength, $\mathbf{B} = \mu_0 \mathbf{H} + \mathbf{M}$ (μW/mm^2) is the magnetic induction with $\mathbf{M}$

(μW/mm^2) the magnetisation of the medium and $\mu_0 = 4\pi\ 10^{-1}$ N/mA^2 is the permeability of free space.

Although the physical interpretation of Maxwell's equations is not immediately obvious from equations (2.3)-(2.6), they can be interpreted as follows. Equation (2.3) is the differential form of Gauss's law, which states that the total flux of displacement field out of a closed surface is equal to the net charge within the volume enclosed by the surface. Equation (2.4) asserts that magnetic monopoles do not exist. The second pair of equations relate current and magnetic fields. Equation (2.5) is Ampere's law which states that the current density and displacement field give rise to the magnetic field **H**. Equation (2.6) is Faraday's law of induction which states that a time varying magnetic field gives rise to an electric field. The negative sign is Lenz's law which asserts that energy must be expended to generate this electric field.

Under the conditions of dendritic modelling, it is reasonable to assume that **P**, **M** and **J** are parallel to **E**, **H** and **E** respectively, so that

$$\mathbf{D} = \kappa\epsilon_0\mathbf{E}, \qquad \mathbf{B} = \mu\mu_0\mathbf{H}, \qquad \mathbf{J} = \sigma\mathbf{E} \tag{2.7}$$

where κ is a dielectric constant, μ is a permeability factor and σ is an electrical conductivity. Assuming that κ, μ and σ are constant functions of position, Maxwell's equations can now be written as

$$\text{div } \mathbf{E} = \frac{\rho}{\kappa\epsilon_0}, \tag{2.8}$$

$$\text{div } \mathbf{H} = 0, \tag{2.9}$$

$$\text{curl } \mathbf{H} = \sigma\mathbf{E} + \kappa\epsilon_0\frac{\partial\mathbf{E}}{\partial t}, \tag{2.10}$$

$$\text{curl } \mathbf{E} = -\mu\mu_0\frac{\partial\mathbf{H}}{\partial t}. \tag{2.11}$$

The three-dimensional model for a dendrite is developed from Maxwell's equations by assessing the relative importance of the individual terms in the equations with reference to both the morphology and biophysical properties of the dendrite. For the extra- and intracellular media, it is necessary to choose suitable values for κ, μ and σ with respect to the dendritic medium. The value of $\kappa \approx 81$ is taken to be that of water. The value of $\mu \approx 0.022$ is estimated from the refractive index of water ($n = 1.33$) using the formula $n = \sqrt{\kappa\mu}$. The value for the specific conductivity $\sigma \approx 3.3$ mS/mm is taken from Katz (1966). The typical dimensions for dendritic radius and length are $R \approx 5 \times 10^{-6}$m and

$L \approx 10^{-3}$m, respectively. I first consider the dispersion of electric charge and the diffusion of the magnetic field.

2.3.1 Dispersion of charge

Maxwell's third equation (2.10) governs the dispersion of charge in the intra- and extra-cellular media,

$$\text{curl } \mathbf{H} = \sigma\left(\mathbf{E} + \beta\frac{\partial \mathbf{E}}{\partial t}\right), \qquad \beta = \frac{\kappa\epsilon_0}{\sigma}. \tag{2.12}$$

In the context of dendritic modelling, $\beta \approx 2.2 \times 10^{-10}$ seconds in the intra-cellular region, with a similarly small value in the extra-cellular region. It follows from equations (2.8) and (2.12) that the density of free charge ρ satisfies the partial differential equation

$$\frac{\partial \rho}{\partial t} + \frac{\rho}{\beta} = 0, \tag{2.13}$$

using the vector identity div(curl $\mathbf{H}$) = 0. The general solution of this equation is

$$\rho(\mathbf{r}, t) = \rho(\mathbf{r}, 0)e^{-t/\beta}, \tag{2.14}$$

where $\rho(\mathbf{r}, 0)$ is the initial distribution of charge. From this solution, it is clear that free charge decays with time constant β seconds, which is negligible, compared with changes in transmembrane potential which occur on the order of microseconds. Therefore, the dispersal of free charge has a negligible contribution to current flow and is assumed to be instantaneous for the purposes of dendritic modelling. Furthermore, the time rate of change of $\mathbf{E}$ in equation (2.12) is significant only if it occurs on a time scale of β seconds, and therefore the term $\beta\,\partial\mathbf{E}/\partial t$ is also considered negligible on a microsecond timescale.

Therefore, the first particularisation of Maxwell's equations comes from the biophysical properties of the dendritic material and leads to the result

$$\text{div } \mathbf{E} = 0, \tag{2.15}$$

$$\text{div } \mathbf{H} = 0, \tag{2.16}$$

$$\text{curl } \mathbf{H} = \mathbf{J} = \sigma\mathbf{E}, \tag{2.17}$$

$$\text{curl } \mathbf{E} = -\mu\mu_0\frac{\partial \mathbf{H}}{\partial t}. \tag{2.18}$$

The next stage in the particularisation of Maxwell's equations considers both the neuronal geometry and electrical properties of the dendrite.

2.3.2 Diffusion of magnetic field

Let us now consider the effect of the magnetic field on the electric field, as defined in equations (2.17) and (2.18).

An implicit assumption in neuronal modelling is that the electric field is derivable from a potential function, that is, it satisfies Laplace's equation. However it is clear from equation (2.18) that the electric field is not generally derivable from a potential function due to the time changing magnetic field. The aim then is to find conditions under which the right hand side of equation (2.18) is negligible. Toward this end, $\mathbf{E}$ is eliminated between equations (2.17) and (2.18) to obtain

$$\text{curl curl } \mathbf{H} = -\sigma\mu\mu_0 \frac{\partial \mathbf{H}}{\partial t}. \tag{2.19}$$

Using the vector identity curl curl $\mathbf{H}$ = grad div$\mathbf{H} - \triangle\mathbf{H}$ in combination with div $\mathbf{H} = 0$ equation (2.19) reduces to

$$\triangle \mathbf{H} = \sigma\mu\mu_0 \frac{\partial \mathbf{H}}{\partial t}. \tag{2.20}$$

This expression can be non-dimensionalised using the changes of variable $\widehat{\mathbf{H}} = \mathbf{H}/\mathrm{H}_0$, $\widehat{\mathbf{x}}$ = x/L and $\widehat{t}$ = t/T, giving the non-dimensional expression

$$\widehat{\triangle}\widehat{\mathbf{H}} = \frac{\sigma\mu\mu_0 \mathrm{L}^2}{T} \frac{\partial \widehat{\mathbf{H}}}{\partial \widehat{t}}. \tag{2.21}$$

The magnitude of the non-dimensional parameter $\sigma\mu\mu_0\mathrm{L}^2/T$ governs the impact of the time changing magnetic field on the electric field. For a motor neuron with a soma of diameter $\approx 10^{-4}$ and electrical activity resolved to 10^{-1} ms, this parameter is approximately 10^{-11}. Therefore the right hand side of equation (2.21) is negligible and so the magnetic field is determined by the electric field through Maxwell's equations (2.16) and (2.17) but cannot itself drive the electric field. Under these conditions, the electric field satisfies curl $\mathbf{E} = \mathbf{0}$, and is therefore derivable from a scalar function.

2.3.3 Maxwell's equations in neuronal modelling

Taking into account both the biophysical and geometrical properties of neuronal material suggest that the most appropriate particularisation of Maxwell's equations is based on instantaneous dispersal of free charge and instantaneous diffusion of the magnetic field. Consequently, Maxwell's equations divide into two pairs of equations. The first pair is

$$\text{div } \mathbf{E} = 0, \qquad \text{curl } \mathbf{E} = 0, \tag{2.22}$$

which leads to the result that

$$\mathbf{E} = -\text{grad}\ \phi, \qquad \mathbf{J} = -\sigma\text{grad}\ \phi, \qquad \nabla^2\phi = 0. \tag{2.23}$$

Therefore the electric fields in the intra- and extra-cellular regions are derived from potential functions which satisfy Laplace's equation. The second pair of equations is

$$\text{div}\ \mathbf{H} = 0, \qquad \text{curl}\ \mathbf{H} = \mathbf{J} = \sigma\mathbf{E}, \tag{2.24}$$

which determine the magnetic field $\mathbf{H}$ from the electric field $\mathbf{E}$.

2.3.4 Equations for a three-dimensional axonal cylinder

In assessing the influence that the three-dimensional representation of the axon has on conduction speed, we consider an infinitely long axon of constant radius a, with intracellular fluid of finite conductivity g_A emersed in extra-cellular fluid of finite conductivity g_E separated by a membrane. The axis of the axon is taken to be the z-axis of a system of cylindrical polar coordinates (r, θ, z) where the axonal membrane has equation $r = a$. The intracellular region is defined by $r < a$ and the extracellular region is defined by $r > a$. The azimuthal symmetry present in the model allows θ to be an ignorable variable. Consequently, the potential $\phi(r, z, t)$ in the intracellular region and the potential $\Phi(r, z, t)$ in the extracellular region are required to be finite solutions of Laplace's equation

$$\frac{\partial^2\phi}{\partial r^2} + \frac{1}{r}\frac{\partial\phi}{\partial r} + \frac{\partial^2\phi}{\partial z^2} = 0, \tag{2.25}$$

$$\frac{\partial^2\Phi}{\partial r^2} + \frac{1}{r}\frac{\partial\Phi}{\partial r} + \frac{\partial^2\Phi}{\partial z^2} = 0. \tag{2.26}$$

However, these potentials are not continuous at $r = a$, and it is the size of this discontinuity that defines the transmembrane potential

$$V_M = \phi(a, z, t) - \Phi(a, z, t). \tag{2.27}$$

The presence of this discontinuity in potential induces transmembrane current J_M to flow across the membrane. In the absence of sources of free charge within the membrane, this current must be identical to that predicted by the gradients of the intra- and extracellular potentials normal to the membrane. These conditions give rise to the boundary conditions

$$J_M = -\lim_{r\to a^-}\left(g_A\frac{\partial\phi}{\partial n}\right) = -\lim_{r\to a^+}\left(g_E\frac{\partial\Phi}{\partial n}\right), \tag{2.28}$$

where g_A and g_E are, respectively, the conductivities of the intra- and extra-cellular media and $\mathbf{n}$ is the unit normal to the membrane directed from the intra- to the extra-cellular region.

In dendritic modelling, neither the intra- and extra-cellular potentials nor the transmembrane potentials are known on the dendritic membrane. Instead these functions are determined by requiring continuity of transmembrane current as prescribed in the boundary conditions (2.28) where the functional form of J_M is prescribed by the constitutive formula (2.31).

2.3.5 Identification of the one-dimensional membrane potential

At this stage, it is necessary to define the three-dimensional equation describing the membrane potential of a infinitely long cylindrical axon, derived from Maxwell's equations. The divergence of the Maxwell equation $\mathbf{J} = \text{curl } \mathbf{H}$ gives the identity $\text{div } \mathbf{J} = 0$. The integration of this equation over a volume of axon gives

$$\begin{aligned}\int_{\mathcal{A}(z)\times(a,b)} (\text{div } \mathbf{J})\, r\, dr\, dz = & -\int_{\mathcal{A}(a)} J_z(r,a,t)\, r\, dr + \int_{\mathcal{A}(b)} J_z(r,b,t)\, r\, dr \\ & + \int_{\partial\mathcal{A}(z)\times(a,b)} \mathbf{J}\cdot\mathbf{n}\, R\sqrt{1+R_z^2}\, dz = 0,\end{aligned} \tag{2.29}$$

where the membrane is the surface $r = R(z)$, R_z is the derivative of R with respect to z, J_z is the z-component of $\mathbf{J}$ and the divergence theorem has been used to replace the volume integral on the left by surface integrals on the right. Expression (2.29) is now divided by $2h$ with the choices $a = z - h$ and $b = z + h$, and the limit taken as $h \to 0^+$ to obtain

$$\frac{\partial}{\partial z}\left(\int_{\mathcal{A}(z)} J_z(r,z,t)\, r\, dr\right) + \int_{\partial\mathcal{A}(z)} \mathbf{J}\cdot\mathbf{n}\, R\sqrt{1+R_z^2} = 0 \tag{2.30}$$

where $\mathcal{A}(z)$ is the area formed by the intersection of the axon with a plane of fixed axial coordinate z and $\partial\mathcal{A}(z)$ is the boundary of $\mathcal{A}(z)$. After all integrations are complete, the identity (2.30) contains only z and t.

The biophysical properties of the dendritic material are introduced through a constitutive law for the transmembrane current density $J_M = \mathbf{J}\cdot\mathbf{n}$ (μA/cm^2). There are typically three contributions to J_M, first the density of synaptic current J_{SYN} (μA/cm^2), the density of intrinsic voltage-dependent current J_{IVDC} (μA/cm^2) arising from ionic channels, and the density of capacitive current due to polarisation of the membrane whose lipid bi-layer structure causes it to behave locally like a parallel plate capacitor with plates raised to

the potential difference of the transmembrane potential V_M. The transmembrane current density due to these processes is

$$J_M(V_M) = c_M \frac{\partial V_M}{\partial t} + J_{SYN}(V_M) + J_{IVDC}(V_M), \tag{2.31}$$

where c_M (μF/cm^2) is the specific capacitance of the membrane and V_M is the transmembrane potential defined in (2.27).

The three-dimensionally derived one-dimensional representation of the cylindrical segment given in equation (2.30) can be further simplified by substituting $\boldsymbol{J} \cdot \boldsymbol{n} = J_M$, where J_z has been replaced by its definition in terms of the axial gradient of the intra-cellular potential $\phi(r, z, t)$. This gives

$$-\frac{\partial}{\partial z}\left(A(z) \int_0^{R(z)} \sigma_A \frac{\partial \phi\,(r, z, t)}{\partial z}\, r\, dr \right) + P(z)\, J_M(V_M) = 0 \tag{2.32}$$

where $V_M = \phi_A - \Phi_E$ is the transmembrane potential. The membrane potential, V_M, derived via Maxwell's equation is now compared with the membrane potential, $\mathcal{V}$, given by the conventional one-dimensional cable equation

$$-\frac{\partial}{\partial z}\left(\sigma_A A(z) \frac{\partial \mathcal{V}}{\partial z} \right) + P(z)\, J_M(\mathcal{V}) = 0, \tag{2.33}$$

where $J_M(\mathcal{V})$ is the transmembrane current density at membrane potential $\mathcal{V}$ as defined in equation (2.31) and $P(z) = 2\pi R(z)$ and $A(z) = \pi R^2(z)$.

The question now is to determine under what conditions V_M is identical to $\mathcal{V}$. To achieve this objective it is necessary to compare the expressions for transmembrane current and the diffusion of axial current between equation (2.32) and equation (2.33).

Comparison of transmembrane current

The reconciliation of $\mathcal{V}$ with V_M requires that the membrane currents from both equations are equivalent, that is,

$$P(z)\, J_M(\mathcal{V}) \equiv P(z)\, J_M(V_M), \tag{2.34}$$

where J_M is defined in equation (2.31). As the transmembrane current $J_M(V_M)$ is the sum of the independent components described in equation (2.31), then a satisfactory definition

of $\mathcal{V}$ must satisfy each of the equations

$$\begin{aligned} P(z)c_M\mathcal{V} &\equiv P(z)c_M V_M, \\ P(z)J_{IVDC}(\mathcal{V}) &\equiv P(z)J_{IVDC}(V_M), \\ P(z)J_{SYN}(\mathcal{V}) &\equiv P(z)J_{SYN}(V_M), \end{aligned} \tag{2.35}$$

because each current is an independent entity. The first condition of (2.35) requires that $\mathcal{V} = V_M$, and if this is the case, then the second and third conditions of (2.35) are satisfied automatically. That is, in the presence of strong-cylindrical symmetry the three-dimensionally derived one-dimensional transmembrane potential V_M and the one-dimensionally derived one-dimensional transmembrane potential $\mathcal{V}$ are required to be identical with respect to transmembrane current.

Comparison of diffusion of axial current

The reconciliation of $\mathcal{V}$ and V_M further requires a comparison of the terms in equations (2.32) and (2.33) which represent the diffusion of axial current. To facilitate this comparison, it is convenient to start with the mathematical identity

$$\int_0^{R(z)} \sigma_A A(z)\frac{\partial \phi}{\partial z}\, r\, dr = \sigma_A A(z)\frac{\partial V_M}{\partial z} + \int_0^{R(z)} \sigma_A \frac{\partial(\phi - \phi_A)}{\partial z}\, r\, dr + \sigma_A A(z)\frac{\partial \Phi_E}{\partial z} \tag{2.36}$$

where V_M and Φ_E are defined on the membrane and are therefore independent of r. This identity allows the partition of the potential derived from Maxwell's equations into three components. The first term on the right hand side of this identity represents the diffusion of axial current in the conventional one-dimensional cable equation. The second term represents the discrepancy in intracellular axial current when the true intracellular potential is represented by its value at the inner boundary of the membrane. Finally the third term is a correction to the intracellular axial current arising from axial variation of the extracellular potential on the outer boundary of the membrane. To complete the correspondence between equations (2.32) and (2.33), the diffusion term in equation (2.32), when expressed using the right-hand side of equation (2.36), must be associated with the diffusion term of equation (2.33). Therefore

$$\sigma_A A(z)\frac{\partial \mathcal{V}}{\partial z} = \sigma_A A(z)\frac{\partial V_M}{\partial z} + \int_0^{R(z)} \sigma_A \frac{\partial(\phi - \phi_A)}{\partial z}\, r\, dr + \sigma_A A(z)\frac{\partial \Phi_E}{\partial z}. \tag{2.37}$$

It is clear from equation (2.37) that the reconciliation between $\mathcal{V}$ and V_M requires that

$$\int_0^{R(z)} \sigma_A \frac{\partial(\phi - \phi_A)}{\partial z}\, r\, dr + \sigma_A A(z)\frac{\partial \Phi_E}{\partial z} = 0. \tag{2.38}$$

In the development of the conventional cable equation, it is implicitly assumed that these terms are negligible. They represent irreducible components in the description of the membrane potential that have no representation in the traditional one-dimensional cable equation. This expression has been derived for a cylindrical uniform axon without taper and implies that the introduction of taper may result in a greater difference between the conventional one-dimensional cable equation and the three-dimensionally derived one-dimensional model.

The identification of terms in the one-dimensional cable equation derived from Maxwell's equations which cannot be eliminated may have far-reaching implications for models of axons or dendrites based on the conventional one-dimensional derivation of the cable equation. In particular, Hodgkin and Huxley developed a membrane model from the membrane kinetics observed in voltage clamp studies on the squid giant axon. A strong test of the model was its ability to predict the conduction speed of a propagated action potential. In this context, two important assumptions need to be examined, the first is the adequacy of the representation of the three-dimensional axon by a one-dimensional model and the second is the validity of the assumption that the travelling wave speed is attained experimentally. We treat the former first and consider the latter in Section 2.5.

2.3.6 Three-dimensional axon with Hodgkin-Huxley kinetics

Axons are conventionally modelled as cylinders. In the case of Hodgkin and Huxley (1952d), the axon is assumed to have a constant radius a (cm) and the Hodgkin-Huxley membrane J_M is defined to be

$$J_M = c_M \frac{\partial V_M}{\partial t} + g_{Na}\, m^3 h\, (V_M - V_{Na}) + g_K\, n^4\, (V_M - V_K) + g_L\, (V_M - V_L) \qquad (2.39)$$

where c_M is the membrane capacitance, g_{Na}, g_K and g_L are respectively the sodium, potassium and leakage conductances (mS/cm^2), V_{Na}, V_K and V_L are respectively the sodium, potassium and leakage equilibrium potentials (mV) and V_M is the transmembrane potential (mV). The auxiliary functions h, m and n define the kinetic behaviour of the conductances through differential equations of the form

$$\frac{dy}{dt} = \alpha_y\, (1 - y) - \beta_y\, y \qquad (2.40)$$

where $y = h, m, n$ and α and β are functions of V_M. Each function is in effect an activation probability, determined experimentally from data based on a number of axons.

The boundary conditions (2.28) written in terms of the Hodgkin-Huxley membrane (2.39) are

$$\begin{aligned} -g_E \frac{\partial \Phi}{\partial a} &= c_M \frac{\partial V_M}{\partial t} + g_{Na}\, m^3 h\, (V_M - V_{Na}) \\ &\quad + g_K\, n^4\, (V_M - V_K) + g_L\, (V_M - V_L) = -g_A \frac{\partial \phi}{\partial a}. \end{aligned} \tag{2.41}$$

Solution procedure

The analysis begins by representing the interior and exterior potentials as the finite Fourier series

$$\phi(r, z; t) = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \phi_k(r; t)\, e^{2\pi i k z/L}, \tag{2.42}$$

$$\Phi(r, z; t) = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \Phi_k(r; t)\, e^{2\pi i k z/L}. \tag{2.43}$$

In this representation, N is the number of intervals into which the region $(0, L)$ is uniformly subdivided and L is the spatial periodicity of $\phi(r, z; t)$ and $\Phi(r, z; t)$ with respect to the z coordinate. Since ϕ and Φ must be solutions of Laplace's equation, it can be demonstrated that ϕ_k and Φ_k are solutions of the ordinary differential equations

$$\frac{d^2\phi_k}{dr^2} + \frac{1}{r}\frac{d\phi_k}{dr} - \nu_k^2 \phi_k = 0, \qquad \nu_k = \frac{2\pi k}{L}, \tag{2.44}$$

$$\frac{d^2\Phi_k}{dr^2} + \frac{1}{r}\frac{d\Phi_k}{dr} - \nu_k^2 \Phi_k = 0, \qquad \nu_k = \frac{2\pi k}{L}, \tag{2.45}$$

for the interior and exterior regions respectively. The general solution of these equations takes the form

$$\phi_k = A_k I_0(\nu_k r) + B_k K_0(\nu_k r) \tag{2.46}$$

where I_0 and K_0 are the modified Bessel functions of the third and fourth kind and A_k and B_k are constants. The requirement that ϕ_k is finite at $r = 0$ is satisfied by the choice $B_k = 0$ and the requirement that Φ_k is finite as $r \to \infty$ is satisfied by the choice $A_k = 0$. Thus the expressions for the potentials ϕ and Φ become

$$\begin{aligned} \phi(r, z, t) &= \sum_{k=-N/2}^{N/2-1} a_k(t)\, I_0(\nu_k r)\, e^{i\nu_k z}, \quad r < a, \\ \Phi(r, z, t) &= \sum_{k=-N/2}^{N/2-1} b_k(t)\, K_0(\nu_k r)\, e^{i\nu_k z}, \quad r > a, \end{aligned} \qquad \text{where } \nu_k = \frac{2\pi k}{L}$$

The coefficients a_k and b_k are directly related through the conservation of charge requirement. The membrane potential can be expressed in terms of a_k and b_k and the specification of membrane current leads to a family of ordinary differential equations for these coefficients. The boundary conditions on the axonal membrane $r = a$ give N first order ordinary differential equations and N algebraic equations for the time course of a_k and b_k. The Hodgkin-Huxley equations (2.40) for each of h, m and n give a further $3N$ first order ordinary differential equations giving $4N$ ordinary differential equations in total. In the absence of injected current, the functions h, m, n and the intracellular and extracellular potentials are constant functions of time. Action potentials in the model are generated by raising the intracellular potential above threshold over a small section in the centre of the axon and integrating the $4N$ differential equations forward in time by numerical integration.

Conduction velocity in a three-dimensional axon

The move to the three-dimensional model gives a slight decrease in the conduction speed from 18.73 ms^{-1} to 18.61 ms^{-1}, a difference of 0.12 ms^{-1}. Thus the approximation of the three-dimensional axon by the traditional cable equation does not account for the observed discrepancy between the predicted and observed speed of the propagated action potential (in fact, the traditional cable equation is over-optimistic). However, in other applications using a one-dimensional model, for example predicting latencies, this difference may be significant.

2.4 Measurement error and biological variability

It is clear from the preceding section, that the discrepancy between theory and experiment cannot be explained by the use of a one-dimensional model to predict the speed of the propagated action potential. Other possible sources of error are now investigated.

Errors associated with the experimental estimation of model parameters from an axon and the errors involved in deriving the rate functions have to be considered as possible sources of error. The former is errors in the measurement of parameters from the axon, for example, conductance or capacitance, while the latter is biological variability due to the derivation of the rate functions from data collected from several axons. The influence of the two forms of error on the predicted and observed conduction speed will be treated separately in this analysis.

To test the influence of possible errors in parameter estimation and biological variability on the conduction speed of the propagated action potential, three distinct simulation exercises are carried out. Each simulation exercise is based on 2000 calculations of the conduction speed. The values of the estimated parameters are drawn from distributions of the parameter values based on the experimental work of Hodgkin and Huxley. Therefore each simulation is considered to give a measurement of the conduction speed for a single axon. The first simulation exercise assesses the influence of measurement error on the theoretical conduction speed of the propagated action potential, whereas the second and third assess the combined effect of measurement error and biological variability.

2.4.1 Measurement error

The mean value and standard deviation of the parameter values used to investigate the influence of measurement error on the conduction speed, are given in Table (2.1). The Hodgkin-Huxley membrane model and definitions of the parameters in Table 2.1 are described in Section 2.3.6.

Parameter	Mean ± Std Dev	Units	Reason
V_{Na}	-72.0 ± 1.00	mV	Double HH error
V_K	55.0 ± 2.00	mV	Double HH error
g_{Na}	120.0 ± 6.00	mS/cm^2	5% relative error
g_K	36.0 ± 1.80	mS/cm^2	5% relative error
g_L	0.3 ± 0.02	mS/cm^2	5% relative error
g_A	28.99 ± 1.45	mS/cm	5% relative error
c_M	1.0 ± 0.05	μF/cm^2	5% relative error
d	476 ± 19.0	μm	4% relative error

Table 2.1: Mean and standard deviation of parameter values for Hodgkin-Huxley membrane model. The final column describes the reason behind each choice of standard deviation.

The mean parameter values in Table 2.1 were taken from the "Value chosen" column of Table 3 in Hodgkin and Huxley (1952d), with the exception of the axonal diameter. The standard deviations of the sodium and potassium equilibrium potentials were chosen to be twice the absolute errors reported by Hodgkin and Huxley (1952a,b). The standard deviations of the remaining parameters were not reported by Hodgkin and Huxley and therefore are given a standard deviation of 5% of the mean value chosen. Figure 1 of Hodgkin and Huxley (1939) allows one to estimate the maximum error in measuring axonal diameter to be approximately 17 μm. The mean axonal diameter reported in the calculation of conduction speed was 476 μm, and so the axonal diameter will be given a standard deviation of 4% (19/476×100 %) in the simulation study. The final parameter to be assigned is the leakage equilibrium potential. Once the parameters in Table 2.1 have been assigned at the start of each simulation, the leakage equilibrium potential is chosen to give a resting membrane potential of -60 mV. Therefore this potential acts like a random variable with each simulation providing a realisation of its value. The computed range of leakage equilibrium potentials can then be compared with the reported range (-56mV to -38mV) in Table 3 of Hodgkin and Huxley (1952d). This provides an internal check on the choice of measurement errors not reported by Hodgkin and Huxley. If the distribution of computed leakage equilibrium potentials corresponds well with the reported range, then the choices of values for the measurement errors are reasonable. For example, if the distribution of leakage equilibrium potentials from the simulation exceeds the reported range, then it is

clear that the magnitude of measurement error and biological variability must be reduced.

2.4.2 Biological variability

The rate functions α_h, β_h, α_m, β_m, α_n and β_n were derived from data collected from several different axons, and therefore it is reasonable to assume that the rate functions will be subject to biological variability. The experimental results for the rate constants α_m, β_m, α_n, β_n, α_h and β_h plotted in Figures 4, 7 and 9 respectively in Hodgkin and Huxley (1952d) imply that the estimation error increases with the value of the function. When the rate functions are small, their observed values lie close to or on the fitted line, suggesting that biological variability can be ignored in this region. As the rate functions increase in value, their spread about the fitted line also increases. However, this increasing variability is only present over the short time-interval for which the membrane potential is distant from its equilibrium value. To reflect the increasing variability of a rate function as its value increases, rate functions in the simulation exercise are calculated by multiplying their Hodgkin-Huxley specification by a Gaussian deviate with mean value one and standard deviation chosen to mimic the largest variability of the data from which that rate function was estimated.

It can be shown for a small sample drawn from a Gaussian distribution that the range is nearly as efficient as the sample standard deviation as a measure of spread in the population (Hoel, 1954). If one defines a standardised range as the ratio of the observed range to the population standard deviation, then Table 2.2 gives the expected value if this ratio for different sample sizes of a Gaussian distributed random variable (Hoel, 1954).

Sample size N	2	3	4	5	6	7	8
E(Range/σ)	1.128	1.693	2.059	2.326	2.534	2.704	2.847

Table 2.2: The expected value of the standardised range of a Gaussian distributed sample of N independent deviates from $N = 2$ to $N = 8$ (see Hoel, 1954).

For this data set, the size of the sample is the number of measurements of the rate function in the close proximity of a given potential, and the range is the maximum relative error in the determination of the rate function of that potential. This information, in combination with the values in Table 2.2, gives a direct estimate of the population standard deviation of the relative error. Table 2.3 displays the estimated standard deviation for each rate

function to be used in the simulation exercises.

Rate Function ms^{-1}	Sample Size N	Maximum Ratio of Range to Mean	Estimated Std. Deviation of Ratio
α_h	4 (4)	0.063/0.163	0.188 (18.8%)
β_h	8 (6)	0.475/0.988	0.169 (16.9%)
α_m	4 (4)	3.130/7.750	0.196 (19.6%)
β_m	5 (3)	0.375/1.500	0.107 (10.7%)
α_n	5 (5)	0.188/0.875	0.092 (9.2%)
β_n	8 (3)	0.050/0.075	0.234 (23.4%)

Table 2.3: The estimated standard deviations of the relative error in the Hodgkin-Huxley rate functions. The integers in brackets in the second column are the number of different axons from which the sample was constructed.

The estimated error for each function will be based on the largest relative error in that function. Thus the simulations will overestimate the influence of biological variability on the theoretical conduction speed making it more difficult to reject the hypothesis that the discrepancy between the computed and observed speed of the propagated action potential is due to biological variability.

2.4.3 Results of simulation exercises

The distribution of conduction speeds from 2000 simulations in the presence of measurement error alone (dashed line) and two combinations of measurement error and biological variability (solid and dotted lines) can be seen in Figure 2.1.

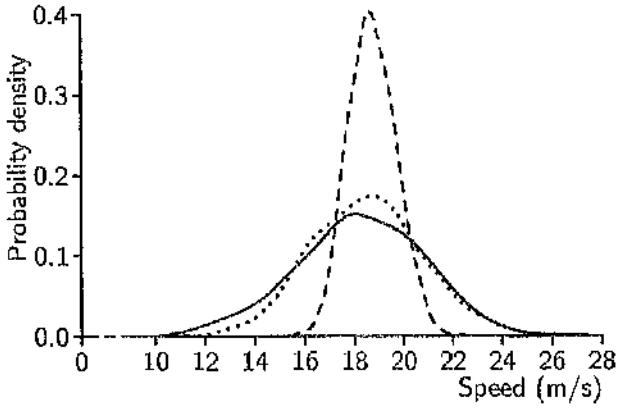

Figure 2.1: Probability density function of the distribution of theoretical conduction speeds based on 2000 simulations of the model axon. The dashed line represents measurement error alone using the standard deviations in Table 2.1, while the solid line incorporates biological variability with measurement error using the errors detailed in Table 2.3. The dotted line is the distribution of conduction speeds with twice the measurement error in Table 2.1 and half the biological variability in Table 2.3.

The first combination of measurement error and biological variability (Figure 2.1, solid line) follows the standard deviations and errors prescribed respectively in Tables 2.1 and 2.3, whereas the second combination (Figure 2.1, dotted line) uses twice the measurement error prescribed in Table 2.1 and half the error associated with biological variability listed in Table 2.3.

For each simulation exercise, the likelihood of obtaining a theoretical conduction speed of at least 21.2 ms^{-1} can be estimated directly from the probability densities[1] shown in Figure 2.1. Therefore, in the presence of measurement error alone, the probability of achieving at least 21.2 ms^{-1} is less than 1 in 200 (0.5%), and approximately 1 in 8 (12-13%) for the combination of measurement error and magnitude of biological variability given in Table 2.3. Given these probabilities, it is unlikely that measurement error alone could account for the 11-12% discrepancy in conduction speed, and the addition of biological variability does not significantly improve this likelihood. In fact, the combinations of measurement error and biological variability used in the simulations skews the distribution of conduction speeds such that there is an increased probability of a slower speed thereby increasing the likely discrepancy between the observed and predicted conduction speeds. Moreover, doubling the measurement error in Table 2.1 and halving the error associated

[1]The procedure for estimating probabilities from probability densities is described in Chapter 5.

with biological variability in Table 2.3 made a negligible difference to the probability of predicting a conduction speed of at least 21.2 ms^{-1}.

An internal check of each simulation exercise lies in the distribution of leakage equilibrium potentials shown in Figure 2.2.

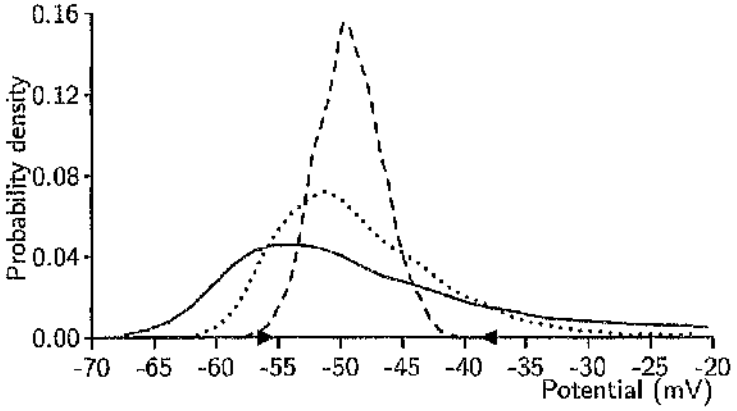

Figure 2.2: Probability density function of the leakage equilibrium potential chosen to maintain an equilibrium membrane potential of -60 mV. The dashed line represents measurement error alone using the standard deviations in Table 2.1, while the solid line incorporates biological variability with measurement error using the errors detailed in Table 2.3. The dotted line is the distribution of conduction speeds with twice the measurement error in Table 2.1 and half the biological variability in Table 2.3. The black inward-pointing arrows indicate the range of leakage equilibrium potentials reported by Hodgkin and Huxley (1952d).

In the first simulation involving measurement error alone (dashed line, Figure 2.2), the distribution of equilibrium potentials corresponds well with the range of -56 mV to -38 mV reported in Table 3 by Hodgkin and Huxley (1952d). However, when the errors associated with biological variability are combined with measurement error (solid line, Figure 2.2) the distribution of leakage equilibrium potentials far exceeds the range reported by Hodgkin and Huxley (1952d).

Although the choice of biological variability detailed in Table 2.3 combined with the measurement error listed in Table 2.1 gives a 1 in 8 probability of predicting a conduction speed of at least 21.2 ms^{-1}, it is clear that the levels of biological variability involved are excessive. Additionally the third simulation exercise, which doubled the measurement error in Table 2.1 and halved the errors in Table 2.3 associated with biological variability, still found that the distribution of leakage equilibrium potentials exceeded the range

reported by Hodgkin and Huxley.

Having eliminated numerical inaccuracy and parameter estimation error as possible sources of the discrepancy between predicted and observed conduction speeds, the structure of the membrane model itself must be investigated. However, in the 50 years since the publication of the model, experimental evidence has continued to support the formulation of the membrane model proposed by Hodgkin and Huxley. The observation that sodium channels are composed of four voltage-sensitive units (Caterall, 1988; Sato, Ueno, Asai, Takahashi, Sato, Engel, and Fujiyoshi, 2001) corresponds well with Hodgkin and Huxley's four-step activation kinetics, and similarly for the configuration of potassium channels (see Kreusch, Pfaffinger, Stevens and Choe, 1998; Meunier and Segev, 2002). There has also been significant work which has led to modifications in the description of the behaviour of both the sodium and potassium channels (Armstrong and Bezanilla, 1977; Bezanilla and Armstrong, 1977; Caterall, 1992; Pallotta and Waggoner, 1992), however the new channel models have not yet been used to predict the conduction speed of the propagated action potential in the squid giant axon.

Thus far we have shown that numerical inaccuracies, the one-dimensional approximation of the three-dimensional axon, measurement error and biological variability cannot account for the discrepancy between observation and theory. In view of recent experimental evidence, we choose to retain the kinetic model proposed by Hodgkin and Huxley. Therefore, one important factor that remains to be investigated as a source of the discrepancy between observation and prediction is the validity of the travelling wave assumption.

2.5 The travelling wave assumption

To reduce the partial differential equations to ordinary differential equations, Hodgkin and Huxley (1952d) assumed that the propagating action potential was a travelling wave, namely a wave of invariant shape moving at constant speed. However, it is possible that the experimental action potential did not achieve the travelling wave speed. If this is the case, then any simulation procedure must correspond with the experimental setup and the theoretical conduction speed must be measured over the same distance as that available experimentally.

Returning to the three-dimensional model, a solution is now required that begins with the axon initially at rest, then following a brief stimulation, an action potential propagates away from the site of stimulation. Experimentally, a rapid injection of current is given at a fixed point on the axon to generate an action potential. This effect is achieved in the model by raising the membrane potential above threshold over a small length of axon.

The solution of the three-dimensional model for the time course of the membrane potential at intervals of 0.1ms after stimulation of the axon can be seen in Figures 2.3A.

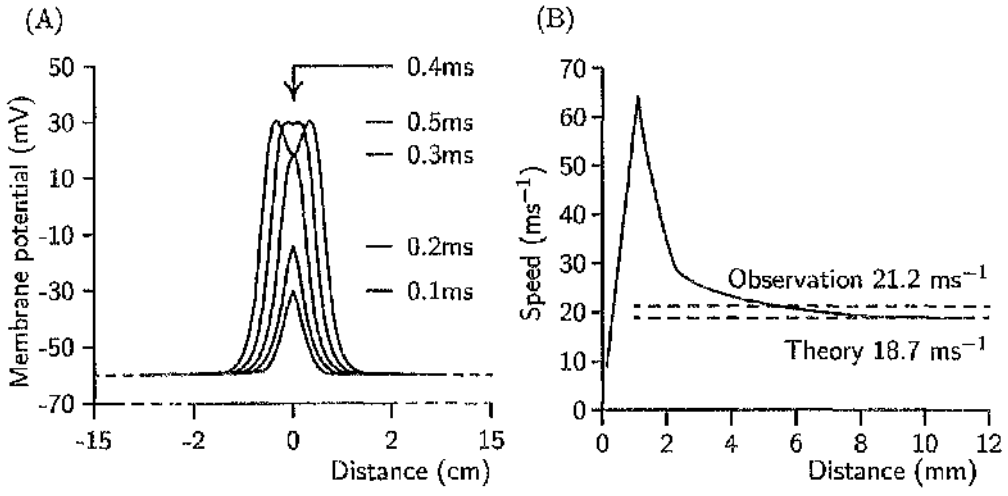

Figure 2.3: Development of the computed action potential and its speed. (A) shows the spatial distribution of the computed axonal membrane potential at times 0.1 ms, 0.2 ms, 0.3 ms, 0.4 ms and 0.5 ms after stimulation of a small section of axon. (B) shows the speed of the peak of the computed action potential versus distance travelled from the point of stimulation. The upper and lower dashed lines in (B) refer to the reported conduction speed and the computed travelling wave speed respectively.

The membrane potential grows in the first 0.3ms until the peaks of two action potentials can be seen at 0.4ms and are clearly defined propagating away from the point of stimulation after 0.5ms. The first discernable peak at 0.4ms is taken as the starting point for the measurement of the conduction speed. The speed is calculated by measuring the distance travelled by the peak of the action potential over intervals of duration 0.01ms. The conduction speed of the action potential as it propagates away from the point of simulation is illustrated in Figure 2.3B. It can be seen that the computed action potential attains the travelling wave (or steady state) speed only after it has travelled at least 9mm away from the point of stimulation. Therefore, a valid comparison of the conduction speeds of the observed and predicted action potentials requires the stimulating and recording electrodes to be at least 9mm apart, although in practice the temporal resolution of the recording equipment may require a greater distance.

In Figure 8B, Miller and Rinzel (1981) plot the instantaneous speed of the 'pulse upstroke' against the distance travelled by the propagated action potential in response to a stimulus. Clearly evident in this figure is an initial transient increase in the conduction speed of the action potential before it settles down to its steady state speed. Miller and Rinzel do not comment on this effect. This transient increase in speed is very similar to that illustrated in Figure 2.3B.

Unfortunately, Hodgkin and Huxley do not describe the experimental conditions under which the conduction speed of the propagated action potential was recorded from the axon. If the recording chamber drawn in Figure 1 from Hodgkin *et al.* (1952) is assumed to be that used to measure the conduction speed, then the dimensions of the chamber suggest that it may not be appropriate to take the travelling wave speed as the predicted conduction speed. Furthermore, the transient effect of stimulation generates an action potential which initially moves much faster than the steady state speed. This implies that the discrepancy between the observed and predicted conduction speeds may be resolved if the stimulating and recording electrodes are suitably close. Of course, the size of the discrepancy will depend critically on the experimental apparatus, the point of stimulation and the strength of the stimulation used to generate the action potential. The transient behaviour of the action potential will now be further investigated.

2.5.1 A possible explanation for the discrepancy between the observed and predicted conduction speed

To understand how the discrepancy between the observed and predicted conduction speeds can occur, the shape of the action potential as it moves away from the point of stimulation is examined. Figure 2.4A shows the speed of the leading edge of the action potential at selected values of the membrane potential.

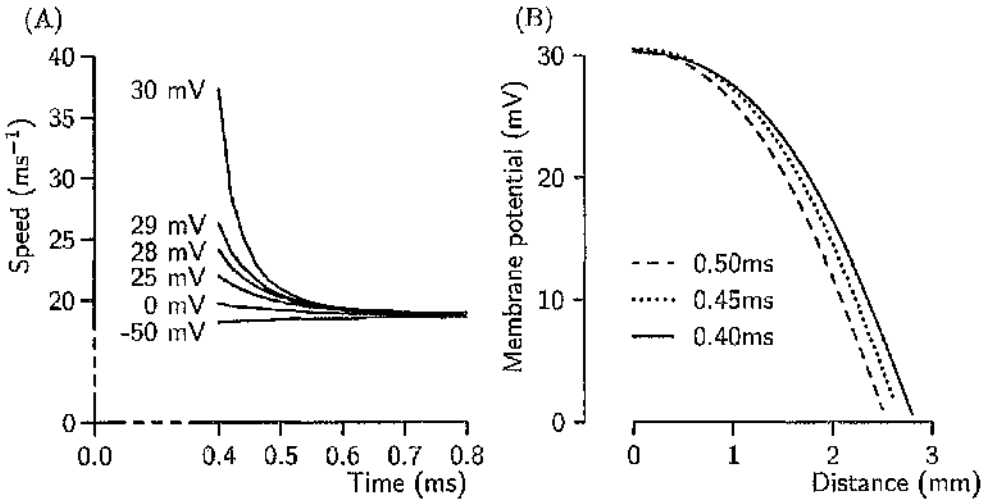

Figure 2.4: Changing shape and speed of the computed action potential. (A) shows the speed of the leading edge of the computed action potential at selected values of the membrane potential. (B) illustrates the changing shape of the leading edge of the computed action potential in the time interval 0.4 ms to 0.5 ms after stimulation.

It is clear from Figure 2.4A that the peak of the action potential is travelling almost twice as fast as its base at 0.4 ms after stimulation. It takes an additional $0.3 - 0.4$ ms for all parts of the action potential to reach a steady state speed, by which time the action potential has travelled approximately 9mm away from the point of stimulation. Aligning the peaks of the action potentials recorded at 0.4 ms (solid line), 0.45 ms (dotted line) and 0.5 ms (dashed line) in Figure 2.4B reveals that the leading edge of the action potential steepens as it moves away from the point of stimulation. It appears that this change in shape, although small, can account for the transient increase in speed of the action potential prior to achieving the travelling wave speed.

2.6 Trains of action potentials

An action potential, or a spike, rarely propagates along an axon on its own, but rather is part of a train of action potentials carrying a signal to the next neuron in the pathway. The essential feature of the solution of the three-dimensional model is its assumed periodicity. Within this assumption, it is a simple adjustment to consider trains of action potentials with variable intervals of spatial periodicity.

In the previous section, the length of the spatial repeat pattern was chosen to be deliberately large so that the generation and propagation of a single propagated action potential could be studied. This procedure is adapted to investigate spike trains by adjusting L, the length of the spatial periodicity, to provide a means by which the relationship between the conduction speed and spatial periodicity of the spike train can be quantified.

By changing L, the effect of refractoriness on the amplitude and conduction speed of a spike train can be examined. For example, the spike amplitude reduces as L becomes smaller until a critical value of L is passed beyond which a spike train is not sustainable. Conversely at larger values of L the spikes behave independently and propagate at the conduction speed predicted by Hodgkin and Huxley.

Miller and Rinzel (1981) investigated the dispersive properties of the propagating action potential at a range of temperatures for the Hodgkin-Huxley model by assuming "a periodic train of uniformly spaced pulses travelling with fixed speed". Using a boundary condition problem, the initial conditions corresponded to a time dependent stimulating current that initiated the propagating action potential. They found a range of frequencies at which the propagating action potentials achieved conduction speeds greater than the steady state travelling wave speed. Increasing temperature caused a significant increase in the conduction speed, due primarily to accelerating the recovery process and decreasing the refractory period (Miller and Rinzel, 1981).

2.6.1 The dispersive relationship

The propagated action potential starts with the profile of a travelling wave rather being initiated by an injected current. This wave travels for 20 ms to allow the steady state speed for that choice of L to be achieved before the conduction speed is calculated. The simulation calculates the conduction speed of an action potential in the axon at 18.5°C

with values of L ranging from 0.25 cm to 30.0 cm in increments of 0.25 cm. The conduction speed is calculated by measuring the distance travelled by the peak of the action potential in a given time interval. Figure 2.5A illustrates the profile of the conduction speed and Figure 2.5B illustrates the maximum value of the action potential, both plotted against L.

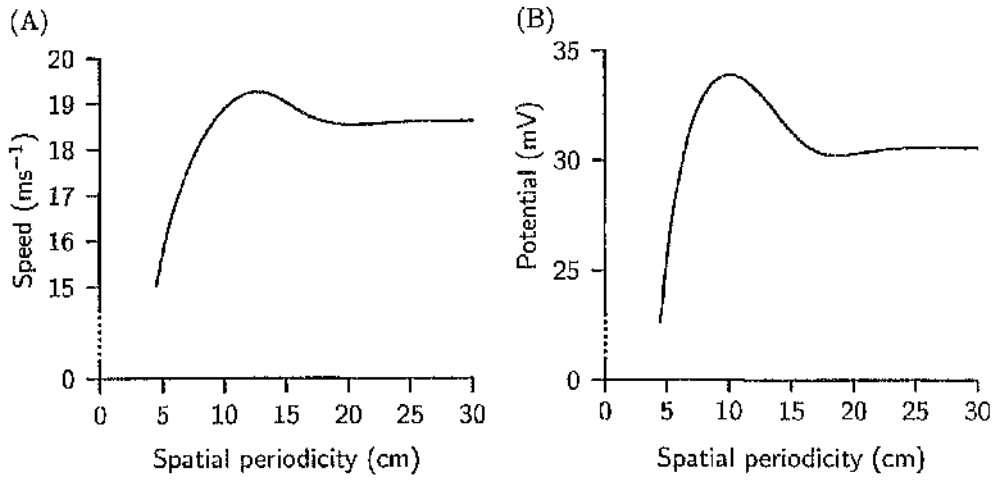

Figure 2.5: Panel (A) shows the conduction speed of the propagated action potential versus the spatial periodicity L, and Panel (B) shows the maximum value of the action potential against the spatial periodicity L.

It is clear from Figure 2.5A that the interval between spikes in a train of action potentials has a significant effect on the conduction speed of that train. Below $L = 4.5$ cm, the train of action potentials cannot be sustained. Above 4.5 cm conduction speed increases monotonically with L until achieving a maximum speed of 19.2 ms^{-1} when $L = 12.5$ cm. Beyond this critical value of L, the conduction speed decreases asymptotically to a steady state speed of 18.6 ms^{-1} - the velocity predicted by the Hodgkin-Huxley model under the assumption of a travelling wave, and is negligibly different from the Hodgkin-Huxley conduction speed when $L > 20$ cm. It is clear from Figure 2.5B that the peak potential also varies with L achieving its maximum value when $L \approx 10.25$ cm. Thus the spike train with the maximum size of the action potential occurs for a spatial periodicity shorter than the spatial periodicity for which the action potential itself has maximum conduction speed. These simulation results agree with Miller and Rinzel (1981) who noted that the peak amplitude occurred at a "somewhat higher frequency" than the peak conduction speed, providing qualitative agreement with our results.

To visualise how the profile of the repeat pattern depends on L, a sample of two repeat patterns for trains with $L = 5\,\text{cm}$ (small), $L = 13\,\text{cm}$ (medium) and $L = 30\,\text{cm}$ (large) are plotted in Figure 2.6A-C.

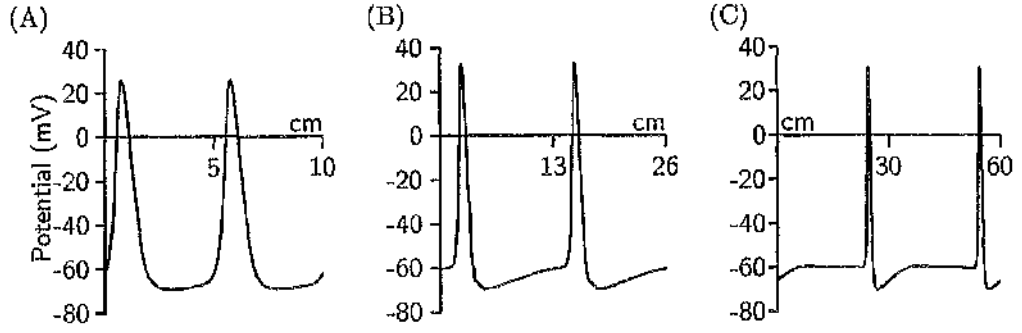

Figure 2.6: Three example spike trains are illustrated for (A) $L = 5\,\text{cm}$, (B) $L = 13\,\text{cm}$ and (C) $L = 30\,\text{cm}$.

The fundamental difference in the three patterns lies in the shape of the recovery period, defined to be the region in which the membrane potential is below -60 mV. When $L = 30\,\text{cm}$ (Figure 2.6C) the individual action potentials are clearly isolated by regions of equilibrium membrane potential (-60 mV). However when $L = 13\,\text{cm}$ and below (Figure 2.6A,B), the membrane potential only momentarily takes the equilibrium potential of -60 mV and the individual action potentials in the train of action potentials are clearly interacting with each other. The question to be addressed now is whether or not the shape of the action potential depends on the spatial periodicity L. Figure 2.7 illustrates the profile of the action potential for $L = 5\,\text{cm}$, $L = 13\,\text{cm}$ and $L = 30\,\text{cm}$ on the same spatial scale.

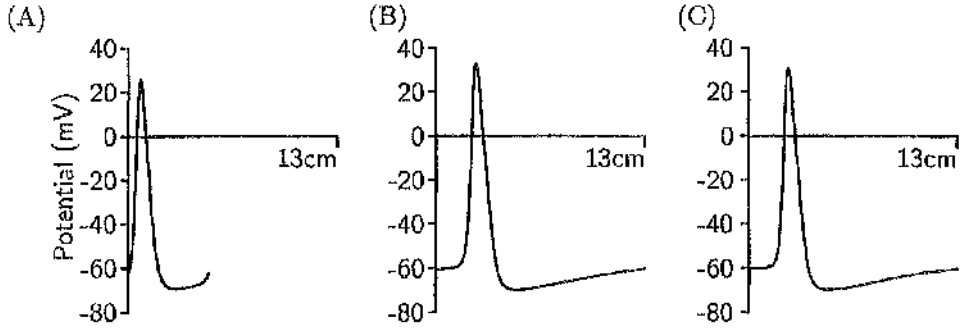

Figure 2.7: The spike trains from Figure 2.6 redrawn on the same scale, where (A) $L = 5\,\text{cm}$, (B) $L = 13\,\text{cm}$, (C) $L = 30\,\text{cm}$.

The conclusions from this figure is that the shape of the action potential is largely inde-

pendent of the choice of L, any dependence enters through the fact that the peak of the action potential is indirectly dependent on L through the duration of the recovery period.

Taking together Figures 2.5 - 2.7 it is clear that the conduction speed and the recovery process depend on L whereas the shape of the action potential is independent of L.

2.6.2 Activation variables

To understand how the processes involved in the formation of an individual action potential manifest themselves in a train of action potentials, the dimensionless auxiliary variables h, m and n are displayed in Figure 2.8A-C for $L = 5\,\text{cm}$, $L = 13\,\text{cm}$ and $L = 30\,\text{cm}$. These variables describe the degree of sodium inactivation, sodium activation and potassium activation respectively.

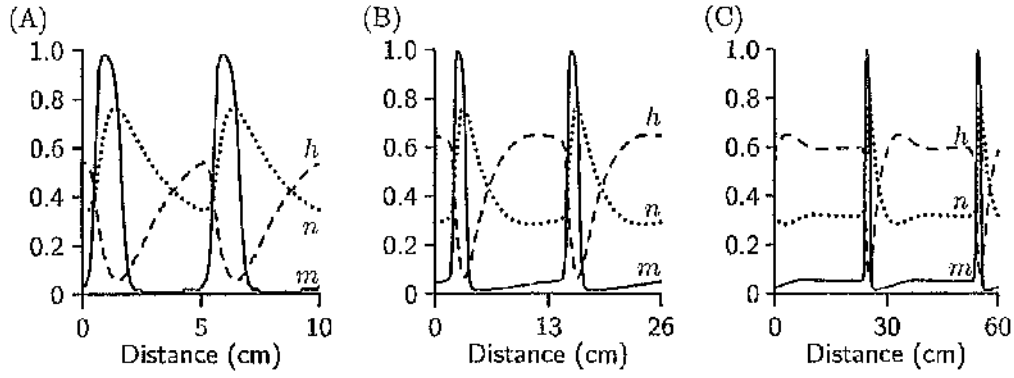

Figure 2.8: The underlying auxiliary variables h, m and n where (A) $L = 5\,\text{cm}$, (B) $L = 13\,\text{cm}$ and (C) $L = 30\,\text{cm}$.

Figure 2.8 suggests that the spatial extent of the sodium activation variable m is essentially independent of the value of L, and in fact this is true. However, the behaviour of the sodium inactivation and potassium activation variables h and n respectively depends on the value of L. When $L = 30\,\text{cm}$ it is clear from Figure 2.8C that all of the auxiliary variables have returned to their equilibrium values. When $L = 13\,\text{cm}$ or below it is clear that h and n do not have the opportunity to return to their equilibrium values, that is, they are unable to complete their recovery profile before the next action potential arrives.

Miller and Rinzel (1981) suggest that the behaviour of the activation variables may explain the elevated conduction speeds at medium values of L. The upstroke of the next action potential occurs at a point in the recovery profile where the potassium activation n is

below and sodium inactivation h is above their respective resting values, therefore the axon may be in a more excitable state than at rest. For large values of L, the activation variables have returned to their resting values and therefore the next action potential will essentially activate a membrane at rest, as if it were the first action potential.

Figure 2.9 displays the profiles of the combined activation variables m^3h and n^4, normalised such that each peak is centred on the origin.

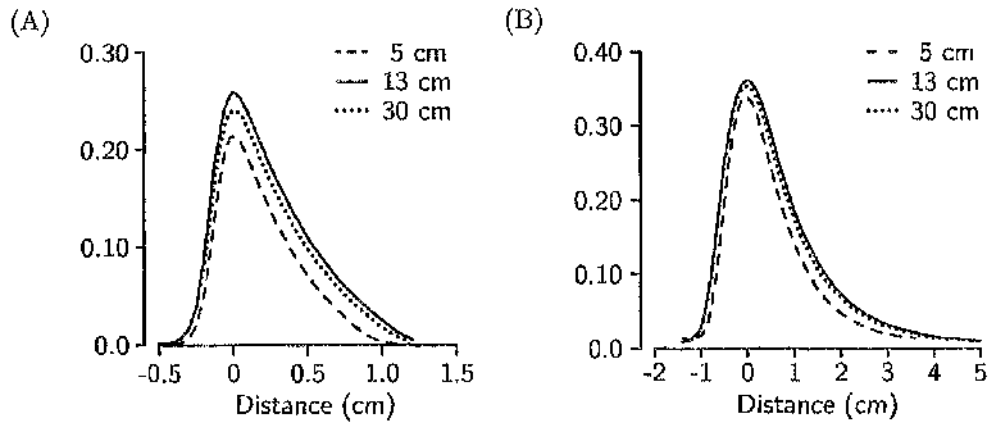

Figure 2.9: Panels (A) and (B) show the profiles of m^3h and n^4 respectively, for $L = 5\,\text{cm}$ (dashed line), $L = 13\,\text{cm}$ (solid line) and $L = 30\,\text{cm}$ (dotted line) normalised such that each peak is centered on the origin.

Consider first Figure 2.9A which illustrates the profiles of m^3h for small, medium and large choices of L. The maximum peak value of m^3h is given by the choice of $L = 13\,\text{cm}$ with the peaks of both $L = 5\,\text{cm}$ and $L = 30\,\text{cm}$ reaching lower values. Conversely in Figure 2.9B the profile of n^4 is essentially unchanged for small, medium and large choices of L. Note that for all values of L, the spatial extent of the m^3h profile is small and has largely completed its cycle by the time n^4 reaches its peak amplitude. The length over which the recovery variable n^4 occurs is approximately 4-5 cm for each value of L. The minimum value of L for a viable train of action potentials is 4.5 cm and is possibly determined by this unchanging variable n^4.

Using the three-dimensionally derived one-dimensional model, the dispersion characteristics of a train of action potentials could be investigated. The conduction speed of the action potentials varied widely with the spatial periodicity L, but settled to a steady speed at large values of L. The period of elevated conduction speeds described by Miller and Rinzel (1981) was present at around 12-13cm, and furthermore the phenomenon of the

maximum peak potential occurring at a slightly smaller value of L than the maximum conduction speed was also evident. Deeper analysis of the underlying activation variables found that the form and conduction speed of an action potential depended critically on the point in the recovery profile at which the action potential was initiated. For a mid-range choice of L, this resulted in a region of elevated conduction speeds, possibly due to an unusual balance of the activation variables h and n which allowed a more potent activation. However, the combined variables m^3h and n^4 were largely unaffected by varying L and instead followed a set response to action potential initiation.

2.7 Conclusions

The derivation of a one-dimensional model from a three-dimensional representation of an axon provided the basis for an investigation into distinct features of the Hodgkin-Huxley membrane model. The three-dimensional model contained irreducible terms that described the behaviour of radial currents in the axon, a feature previously assumed negligible. The application of this model to the Hodgkin-Huxley membrane, in particular, the calculation of the conduction speed of the propagated action potential found only a small difference between the one- and three-dimensionally derived models. In light of this, an explanation for the discrepancy between theory and experiment was investigated. Biological variability and measurement error were both ruled out, and in fact, biological variability was more likely to cause an increase in the discrepancy rather than a decrease.

The determination of the conduction speed of the propagated action potential without making the travelling wave assumption, displays a transient increase in conduction speed before attaining a steady state speed equivalent to that of the travelling wave speed. This analysis has demonstrated that a minimum distance is required to attain the travelling wave speed. However the distance over which Hodgkin and Huxley measured the conduction speed of the propagated action potential remains unknown. Uncharacteristically, they do not provide this information in the article (Hodgkin and Huxley, 1952d), they simply say that the "velocity found experimentally in this fibre was 21.2 m/sec". Without knowing the precise conditions under which this figure was obtained, the discrepancy between the observed and predicted conduction speeds remains.

A recent personal communication with Francisco Bezanilla revealed that it was unlikely that the conduction speed was measured experimentally in the chamber (Hodgkin *et al.* 1952). If this was the case, and the experimental conditions allowed the propagated action potential to attain the travelling wave speed, then the discrepancy cannot be explained by the Hodgkin-Huxley membrane model and the kinetics of the model need to be reconsidered.

Chapter 3

Neuron data and terminology

The procedures to be developed in Chapters 4 and 5 are applied to data gathered from real neurons. To avoid unnecessary repetition, I describe here the anatomical terminology used to define neurons, the experimental procedures used to capture the data and the process by which morphological information is extracted from the raw data.

Terminology

Our classification of dendritic morphology follows Larkman's (1991) description and is illustrated in Figure 3.1. The element of dendrite between the soma and first branch point is called a stem segment, that between a final branch point and dendritic tip is called a terminal segment and that between branch points is called an intermediate segment. In Chapters 4 and 5 a dendritic segment is composed of an arbitrary number of sections.

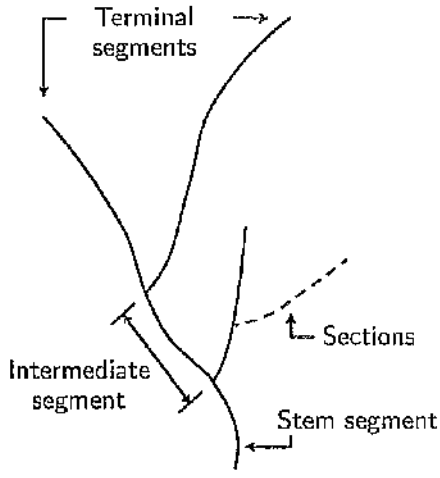

Figure 3.1: An idealised dendrite.

Neuron Data

The morphological data used in Chapters 4 and 5 was collected as part of a larger study investigating whether or not cholinergic spinal interneurons located in laminae III/IV of the dorsal horn of the spinal cord received direct input from primary afferents (Olave, Puri, Kerr and Maxwell, 2002). These interneurons are thought to be last-order interneurons involved in pre-synaptic inhibition (Jankowska, 1992). The cholinergic interneurons were labelled with an antibody raised against choline acetyltransferase (ChaT). The myelinated afferents were labelled with the B-subunit of cholera toxin (CTb), and the unmyelinated afferents were labelled with isolectin B_4 (IB4) and an antibody raised against calcitonin-gene-related peptide (CGRP). All neurons were systematically examined with a BioRad MRC 1024 confocal laser scanning microscope, where sequential images were gathered at $1\,\mu$m intervals from $50\,\mu$m thick vibratome sections. Cells were reconstructed using Neurolucida for Confocal (MicroBrightField, Colchester, VT). Two examples of these interneurons can be seen in Figure 3.2.

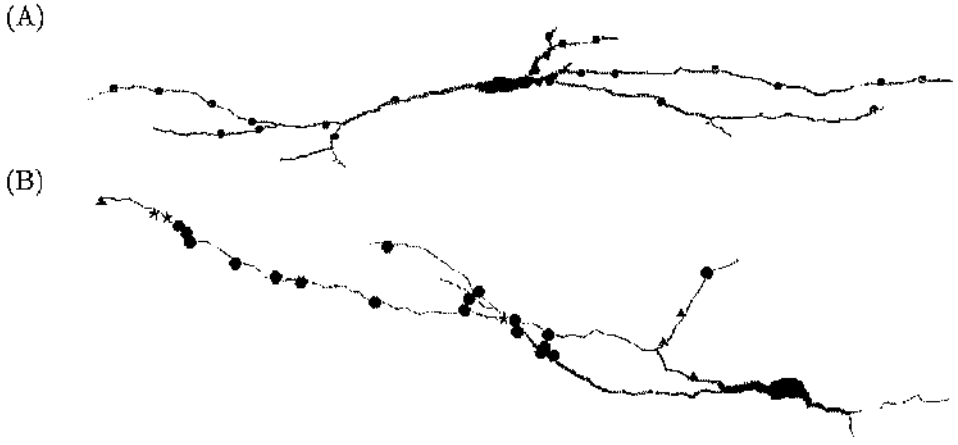

Figure 3.2: Examples of cholinergic interneurons that receive (A) myelinated afferent input and (B) unmyelinated afferent input. The location of the synaptic inputs can be seen on the cells.

The Neurolucida software not only provides information on the location of synaptic contacts, but also provides the Cartesian position, diameter and connectivity pattern for each dendrite (Ascoli, Krichmar, Nasuto and Senft, 2001). The Neurolucida files for my work were kindly provided by David Maxwell.

Extraction of neuron morphology

In a Neurolucida data file a dendritic segment is defined by a sequence of four-vectors in which each four-vector corresponds to a point on the segment. The first three components of the four-vector give the Cartesian coordinates (x, y, z) of the point, and the fourth component is the diameter of the dendritic segment at that point. Let (x_1, y_1, z_1, d_1) and (x_2, y_2, z_2, d_2) be two consecutive points on a dendritic segment, then the length of dendrite between these two points is

$$l = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2},$$

and the associated membrane surface area of this region of dendrite is

$$\text{Surface area} = \frac{\pi(d_1 + d_2)l}{2}$$

on the assumption that this surface area is well approximated by the frustum of a cone.

Connectivity

To facilitate the description of a branched dendrite, it is useful to introduce the notion of parent, child and peer segments. Consider, for example, Figure 3.1. The stem segment has no parent segment and two child segments. The left-hand child segment of the stem segment has the stem segment as parent segment, the right-hand child segment of the stem segment as a peer, and the terminal segments connected at its distal end as child segments. The right-hand child segment of the stem segment has the stem segment as parent, the left-hand child segment of the stem segment as a peer and also has two terminal segments as children. By definition, terminal segments do not have children. Note that it is possible for a segment to be both a stem and terminal segment if it fails to branch before terminating. This is a rare occurrence and is dealt with in Chapter 5.

The overall length of a dendrite is defined to be the sum of the lengths of all its segments. In addition, segments may have synaptic contacts associated with locations (x, y, z) on the dendritic segment. For example, the locations of synaptic contacts are denoted by black circles in Figure 3.2A.

Chapter 4

Analytical development of the equivalent cable

4.1 Introduction

The diverse morphology of dendritic trees has confounded neurophysiologists for over 100 years. Cajal proposed the 'neuron doctrine' which described the neuron, composed of dendritic tree, soma and axon, as the fundamental building block of the nervous system (Cajal, 1952). Furthermore, from the histological tissue sections that he studied under his microscope, he developed the 'principle of dynamic polarisation'. Cajal proposed that the cells received input on the dendritic tree, and that this input was somehow transmitted towards the axon, and from there onto other cells via axodendritic connections. Cajal suggested these ideas from a careful examination of dendritic form using a light microscope and queried how dendrites manage simultaneously many incoming signals.

By contrast with Cajal's view that morphology is important, the predominant view of morphology in the mid-twentieth century based on intracellular recordings from motor neurons was that dendritic structure was unimportant (Jack and Redman, 1995). Rall's first major contribution to neurophysiology was to demonstrate that dendritic morphology was important, and once recognised, then the investigation of dendritic morphology became an important issue. The only tools available to Rall at that time to investigate the function of neuronal morphology was mathematical analysis. Rall's second major contribution to neurophysiology was to show that under certain restrictive conditions a

passive dendritic tree could be described by a single unbranched cable which he called the equivalent cylinder (Rall, 1959).

The core idea in Rall's analysis was to show mathematically that a simple uniform Y-junction is functionally equivalent to a uniform cylinder provided the electrotonic lengths of the two limbs of the Y-junction are identical, and both have the same terminal boundary condition. The Y-junction is equivalent to the Rall cylinder in the sense that for any configuration of input on the branched structure there is an equivalent configuration of input on the equivalent cable such that the electrical behaviour of the Y-junction is indistinguishable from that of the equivalent cylinder at the branch point. Rall showed that the equivalent cylinder had the same terminal boundary condition as the original Y-junction, it had the electrotonic length of one of the limbs of the Y-junction and its conductance was related to that of the limbs of the Y-junction by a 3/2 rule. By this is meant that the sum of the 3/2 powers of the conductances of the limbs of the Y-junction equals the 3/2 power of the conductance of the equivalent cylinder. Finally, Rall showed that currents on the limbs of the Y-junction act on the equivalent cylinder at an electrotonic distance identical to their electrotonic distance on the limbs.

Rall's procedure allows a branched dendrite with uniform segments to be reduced to an equivalent cylinder provided the branched dendrite has the following properties.

1. All terminal boundary conditions are identical, that is, they are all sealed (no axial current flow) or cut (potential held at $V = 0$).

2. The electrotonic length of the dendrite from any branch point to all terminals distal to that branch point is identical.

3. At any branch point, the sum of the 3/2 power of the conductances of all the limbs of the dendrite more distal than the branch point is the 3/2 power of the conductance of the parent limb of the branch point.

To appreciate how these conditions are used in the construction of the equivalent cylinder, the reduction process starts at the dendritic terminals and condenses the outermost branches into equivalent cylinders. This is possible because all terminal limbs have identical boundary conditions and the same electrotonic length to the branch point to which they are attached. Of course, this distance is different for different terminal limbs. Condition 3 guarantees that when the Y-junction is replaced by its equivalent cylinder, this

cylinder has an identical conductance to the parent limb of the branch point. Thus the reduced structure will now have terminal boundary conditions that are identical and uniform limbs. This process can be repeated until the soma of the dendritic tree is reached, and a single equivalent cylinder remains.

Various studies on spinal motoneurons (Barrett and Crill, 1974; Ulfhake and Kellerth, 1983, 1984) have found that dendrites do not conform to the conditions required by Rall for the construction of the equivalent cylinder. To overcome this problem, Clements and Redman (1989) introduced an empirical "equivalent cable" formed by first reducing the limbs of the tree to electrotonic units and then summing the 3/2 powers of the conductances across the tree at the same electrotonic distance from the soma. Note that for a tree obeying the Rall conditions this procedure is exact, and the cable will be a cylinder, but otherwise the new structure will be non-uniform. For dendritic trees that do not follow the Rall conditions, the empirical cable is inaccurate (Whitehead and Rosenberg, 1993) and is not equivalent to the original branched structure.

Furthermore, Whitehead and Rosenberg (1993) demonstrated that equivalent cables could be constructed for branched dendrites which satisfied none of the Rall conditions. This construction was numerical and was based on the Lanczos procedure. The outcome of applying the procedure was a non-uniform cable and a bijective mapping connecting configurations of input on the original dendrite with configurations of input on the cable. One significant disadvantage of the Lanczos procedure was that it suffered from the effects of rounding error (Golub and Van Loan, 1989) and often failed to complete the transformation process in a single operation. Although the numerical procedure demonstrates the existence of equivalent cables for branched structures that do not satisfy Rall's conditions, what is required however is an analytical method to generate equivalent cables, since it is only through an analytical procedure that one can fully understand the equivalent cable.

The aim of this chapter is to describe an exact mathematical procedure by which arbitrary dendrites can be transformed into equivalent cables. In this process, a bijective mapping of configurations of input on the dendritic tree to configurations of input on the equivalent cable is constructed. The procedure will then be applied to real neurons generating unique equivalent cables. For the first time, the locations of synaptic input on real neurons will be mapped to the equivalent cable of that neuron providing an insight into the synaptic distribution.

4.2 Constructing the model dendrite

Constructing a model of a branched dendrite initially appears a daunting task, but may be simplified by treating the dendrite as a collection of segments which are connected such that transmembrane potential is continuous and core current is conserved at branch points. The mathematical model of the dendritic segment forms the elementary unit for constructing a model dendrite, and together with the associated connectivity properties and boundary conditions, forms the complete mathematical model of the branched dendrite.

The mathematical model of the dendritic segment is expressed in terms of the departure of the transmembrane potential $V = V(x,t)$ (mV) from its resting value (assumed to be $V = 0$). The transmembrane potential on a segment with a passive membrane satisfies the *cable equation*

$$P(x)\left(c_M\frac{\partial V}{\partial t} + g_M V\right) + \mathcal{I}(x,t) = \frac{\partial}{\partial x}\left(g_a A(x)\frac{\partial V}{\partial x}\right) \tag{4.1}$$

where $P(x)$ and $A(x)$ are respectively the perimeter and cross-sectional area of the segment at distance x (cm) along the segment, c_M (μF/cm^2) and g_M (mS/cm^2) are the specific capacitance and specific conductance of the segment membrane respectively and g_a (mS/cm) is the conductance of the intracellular medium. The function $\mathcal{I}(x,t)$ (μA/cm) describes the linear density of exogenous transmembrane current and t (ms) measures time. The core current along the segment is calculated from the expression

$$I(x,t) = -g_a A(x)\frac{\partial V(x,t)}{\partial x}. \tag{4.2}$$

At each segment endpoint, the solution of equation (4.1) must maintain conservation of core current and continuity of membrane potential, or satisfy a boundary condition if it is a dendritic terminal.

The next step is to non-dimensionalise the cable equation (4.1) using non-dimensional time s and electrotonic length z given by

$$s = t\frac{g_M}{c_M}, \qquad z = \int_0^x \sqrt{\frac{g_M P(u)}{g_a A(u)}}\,du, \tag{4.3}$$

thereby reducing it to the canonical form. Using the change of variable (4.3), the derivatives in non-dimensional terms are

$$\frac{\partial V}{\partial t} = \frac{\partial V}{\partial s}\frac{ds}{dt} = \frac{\partial V}{\partial s}\frac{g_M}{c_M}, \qquad \frac{\partial V}{\partial x} = \frac{\partial V}{\partial z}\frac{dz}{dx} = \frac{\partial V}{\partial z}\sqrt{\frac{g_M P(x)}{c_M A(x)}}, \tag{4.4}$$

where it is understood in the later expression that occurrences of x are to be replaced by z using the mapping (4.3). Expressions (4.4) can be substituted into the cable equation (4.1) giving

$$P(x)\left(g_M\frac{\partial V}{\partial s}+g_M V\right)+\mathcal{I}=\frac{\partial}{\partial z}\left(g_a A(x)\sqrt{\frac{g_M P(x)}{c_M A(x)}}\frac{\partial V}{\partial z}\right)\sqrt{\frac{g_M P(x)}{c_M A(x)}}, \tag{4.5}$$

which is simplified by dividing through by $g_M P(x)$ to obtain

$$\frac{\partial V(z,s)}{\partial s}+V(z,s)+\frac{\mathcal{I}(x,t)}{g_M P(x)}=\frac{1}{\sqrt{g_a g_M P(x)A(x)}}\frac{\partial}{\partial z}\left(\sqrt{g_a g_M P(x)A(x)}\,\frac{\partial V(z,s)}{\partial z}\right). \tag{4.6}$$

Let the characteristic conductance $c(z)$ of the segment and the non-dimensional current density $J(z,s)$ be defined by

$$c(z)=\sqrt{g_a g_M P(x)A(x)}, \qquad J(z,s)=\mathcal{I}(x,t)\sqrt{\frac{g_a A(x)}{g_M P(x)}}, \tag{4.7}$$

then in terms of these functions, the non-dimensionalised cable equation becomes

$$c(z)\left(\frac{\partial V(z,s)}{\partial s}+V(z,s)\right)+J(z,s)=\frac{\partial}{\partial z}\left(c(z)\frac{\partial V(z,s)}{\partial z}\right). \tag{4.8}$$

The non-dimensionalisation of the transmembrane current is based on the observation that in any time interval $(t, t+dt)$ the charge $\mathcal{I}(x,t)\,dx\,dt$ crossing the membrane occupying $(x, x+dx)$ must equal the charge $J(z,s)\,dz\,dt$ crossing the same portion of membrane now occupying $(z, z+dz)$. The same non-dimensionalisation when applied to the core current (4.2) gives

$$I(z,s)=-g_a A(x)\frac{\partial V}{\partial z}\sqrt{\frac{g_M P(x)}{g_a A(x)}}=-c(z)\frac{\partial V(z,s)}{\partial z} \tag{4.9}$$

where the conductance $c(z)$ is defined in (4.7). The non-dimensional canonical expressions (4.8) and (4.9) define respectively the cable equation and core current for a non-uniform dendritic segment.

Each segment has a cable equation and individual expression for $c(z)$ and $J(z,s)$. To form a branched dendrite requires continuity of membrane potential and conservation of core current at branch points or when connecting to the parent structure. At dendritic terminals, either the transmembrane potential or the core current must be defined. If these requirements are all satisfied, then the mathematical model of the dendritic tree consists of a family of connected cable equations with unique expressions for $c(z)$ and $J(z,s)$.

4.2.1 Mathematical model of a uniform cable

Current input to a dendrite typically comes from synaptic contacts on its membrane. Therefore, in constructing the equivalent cable, current input will be restricted to discrete points spaced uniformly along the electrotonic length of the dendritic segment. The portion of membrane between any two points is called a section, and therefore current input is treated as a boundary condition at section endpoints with $J(z,s) = 0$ on every section. Furthermore, it is assumed that the characteristic conductance $c(z)$ is constant on each section, but a different constant for different sections. Under these conditions, the non-dimensional cable equation (4.4) takes the simplified form

$$\frac{\partial V(z,s)}{\partial s} + V(z,s) = \frac{\partial^2 V(z,s)}{\partial z^2}. \tag{4.10}$$

This procedure, where exogenous membrane current enters the model through the boundary conditions, is similar to that used by Holmes (1986) for the treatment of synaptic input. Holmes (1986) and Van Pelt (1992) used the Laplace transform methodology to develop a continuous cable representation of branched dendrites. The transform variable in this procedure is defined as

$$\widetilde{V} = \int_0^\infty V(z,s)\, e^{-ps} ds. \tag{4.11}$$

The Laplace transformed representation of equation (4.10) becomes

$$\int_0^\infty \frac{\partial V(z,s)}{\partial s}\, e^{-ps} ds + \int_0^\infty V(z,s)\, e^{-ps} ds = \int_0^\infty \frac{\partial^2 V(z,s)}{\partial z^2}\, e^{-ps} ds. \tag{4.12}$$

The first expression on the left-hand side of equation (4.12) gives

$$\begin{aligned} \int_0^\infty \frac{\partial V(z,s)}{\partial s}\, e^{-ps} ds &= \left[V e^{-ps}\right]_0^\infty - \int_0^\infty V(z,s)\,(-p) e^{-ps} ds \\ &= -V(0,s) + p\,\widetilde{V}. \end{aligned} \tag{4.13}$$

The expression on the right-hand side of equation (4.12) gives

$$\int_0^\infty \frac{\partial^2 V(z,s)}{\partial z^2}\, e^{-ps} ds = \frac{d^2}{dz^2}\left(\int_0^\infty V\, e^{-ps} ds\right) = \frac{d^2 \widetilde{V}}{dz^2}. \tag{4.14}$$

Substituting expressions (4.13) and (4.14) into equation (4.12) shows that the Laplace transform of the membrane potential satisfies

$$-V(0,z) + p\widetilde{V} + \widetilde{V} = \frac{d^2 \widetilde{V}}{dz^2}$$

which can be simplified to

$$\frac{d^2\widetilde{V}(z,p)}{dz^2} - \omega^2\widetilde{V}(z,p) = 0, \qquad \omega^2 = p + 1 \tag{4.15}$$

when the initial membrane potential $V(0,z)$ is taken to be zero. The general solution of equation (4.15) is

$$\widetilde{V} = A\,e^{\omega z} + B\,e^{-\omega z}$$

where A and B are arbitrary constants. The equivalent cable analysis is developed from two identities connecting the Laplace transforms of the core currents at either end of a uniform section of length h to the Laplace transforms of the membrane potentials at the section endpoints. This is illustrated in Figure 4.1 for a dendritic section of length h.

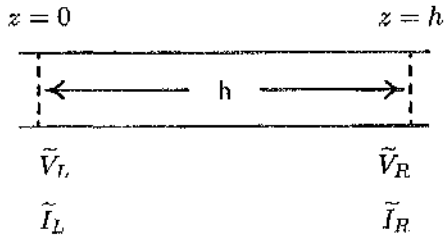

Figure 4.1: A dendritic section of length h, with membrane potential V_L and corresponding current I_L at the left hand end and membrane potential V_R and current I_R at right hand end.

Now consider the dendritic section in Figure 4.1. When $z = 0$, it is clear that $B = \widetilde{V}_L$, and when $z = h$ it is clear that $A\sinh\omega h + B\cosh\omega h = \widetilde{V}_R$, and therefore

$$A\sinh\omega h = \widetilde{V}_R - \widetilde{V}_L\cosh\omega h. \tag{4.16}$$

The potentials $\widetilde{V}_L$ and $\widetilde{V}_R$ must now be connected with the core currents $\widetilde{I}_L$ and $\widetilde{I}_R$. The core current is given by

$$I(z,s) = -c(z)\frac{\partial V(z,s)}{\partial z}$$

where c(z) is the characteristic conductance of the section, and its Laplace transform is

$$\widetilde{I}(z,p) = -c(z)\frac{\partial\widetilde{V}(z,p)}{\partial z} = -c\omega(A\cosh\omega z + B\sinh\omega z)\,.$$

Thus

$$\widetilde{I}(0,p) = \widetilde{I}_L = -c\omega A, \qquad \widetilde{I}(h,p) = \widetilde{I}_R = -c\omega(A\cosh\omega h + B\sinh\omega h).$$

It is now straightforward algebra to demonstrate that

$$\frac{\sinh \omega h}{\omega} \widetilde{I}_L = -c\left(\widetilde{V}_R - \widetilde{V}_L \cosh \omega h\right), \tag{4.17}$$

$$\frac{\sinh \omega h}{\omega} \widetilde{I}_R = -c\left(\widetilde{V}_R - \widetilde{V}_L \cosh \omega h\right) \cosh \omega h - c\widetilde{V}_L \sinh^2 \omega h. \tag{4.18}$$

Using the trigonometric identity $\cosh^2 x - \sinh^2 x = 1$ to simplify expression (4.18), the final identities satisfied by the core current and membrane potential at the left and right section endpoints are

$$\begin{aligned} \frac{\sinh \omega h}{\omega} \widetilde{I}_L &= c\,\widetilde{V}_L \cosh \omega h - c\,\widetilde{V}_R, \\ \frac{\sinh \omega h}{\omega} \widetilde{I}_R &= c\,\widetilde{V}_L - c\,\widetilde{V}_R \cosh \omega h. \end{aligned} \tag{4.19}$$

4.2.2 Model equations for a branched dendrite

The model equations are constructed by requiring continuity of membrane potential and conservation of core current at section endpoints. The equation contributed by a section boundary is formed by equating the exogenous current injected at that boundary to the sum of the core currents from all the segments meeting at that boundary. These boundaries can be either branch points, internal boundaries of a segment, a boundary at a dendritic terminal or the boundary of contact with a parent structure.

Branch point

If a current $I_B(s)$ is injected into a branch point then conservation of core current requires that

$$I_B(s) = I_P(s) - \sum I_C(s), \tag{4.20}$$

where $I_P(s)$ and $I_C(s)$ are the respective core currents in the parent and child segments which meet at the branch point, and summation is taken over *all* child segments. The branch point condition is constructed from the Laplace transform

$$\widetilde{I}_P - \sum \widetilde{I}_C = \widetilde{I}_B \tag{4.21}$$

of equation (4.20). Equations (4.16) for $\widetilde{I}_L$ and $\widetilde{I}_R$ are now particularised to $\widetilde{I}_C$ and $\widetilde{I}_P$ respectively, and therefore the currents in equation (4.21) become

$$\begin{aligned} \frac{\sinh \omega h}{\omega} \widetilde{I}_C &= c^C \widetilde{V}_B \cosh \omega h - c^C \widetilde{V}_C, \\ \frac{\sinh \omega h}{\omega} \widetilde{I}_P &= c^P \widetilde{V}_P - c^P \widetilde{V}_B \cosh \omega h, \end{aligned} \tag{4.22}$$

where $\widetilde{V}_B$ is the Laplace transform of the membrane potential at the branch point, $\widetilde{V}_C$ is the Laplace transform of the membrane potential at the distal end of the first section of a child segment, and $\widetilde{V}_P$ is the Laplace transform of the membrane potential at the proximal end of the last section of the parent segment. Similarly, c^C is the characteristic conductance of the first section of the child segment, while c^P is the characteristic conductance of the last section of the parent segment. Substitution of equations (4.22) into equation (4.21) gives

$$c^P \widetilde{V}_P - \left(c^P + \sum c^C\right)\widetilde{V}_B \cosh \omega h + \sum c^C \widetilde{V}_C = \frac{\sinh \omega h}{\omega} \widetilde{I}_B. \tag{4.23}$$

Again, all summations in equation (4.23) are taken over the child segments. The standardised branch point equation is found by dividing equation (4.23) by the sum of the characteristic conductances of all segments meeting at the branch point, giving

$$\frac{c^P \widetilde{V}_P}{(c^P + \sum c^C)} - \widetilde{V}_B \cosh \omega h + \frac{\sum c^C \widetilde{V}_C}{(c^P + \sum c^C)} = \frac{\sinh \omega h}{\omega(c^P + \sum c^C)} \widetilde{I}_B. \tag{4.24}$$

Contiguous sections

Contiguous sections are treated as a special case of a branch point with a single child segment. For contiguous sections equation (4.24) simplifies to

$$\frac{c^P \widetilde{V}_P}{c^P + c^C} - \widetilde{V}_B \cosh \omega h + \frac{c^C \widetilde{V}_C}{c^P + c^C} = \frac{\sinh \omega h}{\omega(c^P + c^C)} \widetilde{I}_B \tag{4.25}$$

where $\widetilde{V}_B$ is the potential at the section boundary, $\widetilde{V}_P$ is the potential of the proximal end of the left-hand section at the change in diameter and $\widetilde{V}_C$ is the potential at the distal end of the right-hand section.

Connection to parent structure

The equation contributed by the section boundary between the parent structure and the dendritic tree is determined directly from the branch point condition by ignoring all contributions from the parent segment and replacing the injected current I_B by the current I_0 flowing from the dendritic tree into the parent structure. The result is

$$\widetilde{V}_0 \cosh \omega h + \frac{\sum c^C}{\sum c^C} \widetilde{V}_C = \frac{\sinh \omega h}{\omega \sum c^C} \widetilde{I}_0, \tag{4.26}$$

where $\widetilde{V}_0$ is the Laplace transform of the membrane potential at the P_0, the point of connection to the parent structure, and summation takes place over all segments meeting at P_0.

4.3 Notation

The analytical development to follow will apply equations (4.19) to both branched and cable-like structures. Although both will consist of a collection of dendritic segments, the mathematical description of the two is quite different making it necessary to distinguish between equations referring to the cable-like structure and those referring to the branched structure. To avoid confusion, objects relating to the cable will be defined by calligraphic symbols and objects relating to the branched tree structure will be defined by roman symbols. Where no ambiguity exists, a Roman symbol will be used (see Table 4.1).

Description	Cable	Tree	No distinction
Cable matrix	$\mathcal{A}$	A	
Symmetrising matrix	$\mathcal{S}$	S	
Diagonal matrix	$\mathcal{D}$	D	
Tri-diagonal matrix			T
Householder matrix			H
Injected current	$\mathcal{I}$	I	
Membrane voltage	$\mathcal{V}$	V	

Table 4.1: Notation for the matrices and vectors used in the description of the branched and cable-like structures.

4.4 Analytical development of the equivalent cable

Equivalence transformations (Lindsay, Ogden and Rosenberg, 2001a,b) and the Lanczos tri-diagonalisation procedure (Ogden, Rosenberg and Whitehead, 1999) have shown that all equivalent cables take the form of piecewise uniform cables, and therefore represent the canonical form for the equivalent cable. The development of the equivalent cable begins by deriving the mathematical representation of a piecewise uniform cable, and then demonstrating that the mathematical model of an arbitrary branched dendrite with arbitrary input structure may be transformed into a piecewise uniform cable under relatively unrestrictive circumstances.

4.4.1 Construction of the discrete model dendrite

A branched dendrite with n dendritic segments of length $L_1, \ldots, L_n$ is transformed by the non-dimensionalisation defined in (4.3) to a branched dendrite with segments of electrotonic length $l_1, \ldots, l_n$ respectively. The equation for each segment now takes the form of equation (4.4) where segments are defined uniquely by their characteristic conductance $c(z)$ and their electrotonic length.

The construction of the equivalent cable begins by subdividing the dendrite into sections of fixed electrotonic length h. Each segment is assigned an electrotonic length that is the integer multiple of h closest to the segments exact electrotonic length. Any error in the specification of length behaves like a uniformly distributed random variable in the interval $[-h/2, h/2)$. The central limit theorem suggests that the total electrotonic length of the discretised dendrite behaves as a normal deviate with expected value $l_1 + l_2 + \cdots + l_n$ and standard deviation $h\sqrt{n/12}$. Therefore, the electrotonic length of the discretised dendrite can be made arbitrarily close to that of the real dendrite by an appropriate choice of h.

As the electrotonic length of the segment is altered by the discretisation procedure, it is essential to modify $c(z)$ to ensure that the total membrane conductance of a given segment, say the j-th segment, defined by,

$$\int_0^{L_j} P(x) g_M dx = \int_0^{l_j} \sqrt{P(x)A(x) g_M g_a} dz = \int_0^{l_j} c(z) dz,$$

is preserved. To ensure that the discretised piecewise uniform segment and the continuous segment have the same total membrane conductance, the conductance of the k-th section

is assigned the value

$$d_k = \frac{1}{h}\int_{(k-1)l_j/m}^{kl_j/m} c(z)\,dz = \frac{l_j}{mh}\,c\left(\frac{(2k-1)l_j}{2m}\right) + \mathrm{O}(h^2), \qquad k = 1,\ldots,m, \tag{4.27}$$

where mh is the electrotonic length of the discretised segment. The construction of the discretised dendrite from the continuous dendrite is the only approximation made in the development of the equivalent cable, otherwise the analysis is exact.

4.4.2 The piecewise uniform cable

A piecewise uniform cable of electrotonic length nh is shown in Figure 4.2. The cable is divided by the points (or nodes) $P_0, P_1, \ldots, P_n$ into n uniform sections of length h and characteristic conductance d_k. The current $\mathcal{I}_k(s)$ is injected at point P_k at potential $\mathcal{V}_k(s)$.

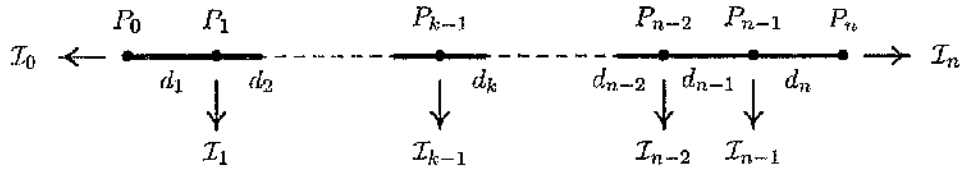

Figure 4.2: A piecewise uniform dendrite with electrotonic length nh, composed of n sections of length h each with characteristic conductance d.

The identities (4.19) are now particularised for the kth section of the piecewise uniform cable to give

$$\frac{\sinh\omega h}{\omega}\,\widetilde{I}_L^{(k)} = d_k\,\widetilde{\mathcal{V}}_{k-1}\cosh\omega h - d_k\,\widetilde{\mathcal{V}}_k, \tag{4.28}$$

$$\frac{\sinh\omega h}{\omega}\,\widetilde{I}_R^{(k)} = d_k\,\widetilde{\mathcal{V}}_{k-1} - d_k\,\widetilde{\mathcal{V}}_k\cosh\omega h, \tag{4.29}$$

where $\widetilde{\mathcal{V}}_k$ is the Laplace transform of the membrane potential $\mathcal{V}_k(s)$ and $\widetilde{I}_L^{(k)}$ and $\widetilde{I}_R^{(k)}$ are respectively the Laplace transforms of the core currents at the left and right hand end-points of the k-th section of the cable. Continuity of membrane potential is guaranteed by construction. The mathematical description of the cable is based on conservation of core current at nodes $P_0, P_1, \ldots, P_n$ and requires that

$$\begin{aligned} I_L^{(1)}(s) &= -\mathcal{I}_0(s), \\ I_R^{(k)}(s) &= -\mathcal{I}_k(s) + I_L^{(k+1)}(s), \qquad (k = 1,\ldots,n-1) \\ I_R^{(n)}(s) &= \mathcal{I}_n(s), \end{aligned} \tag{4.30}$$

or in the Laplace transform space,

$$\widetilde{I}_L^{(1)} = -\widetilde{\mathcal{I}}_0, \quad \widetilde{I}_R^{(k)} = -\widetilde{\mathcal{I}}_k + \widetilde{I}_L^{(k+1)} \quad (k = 1, \ldots, n-1), \quad \widetilde{I}_R^{(n)} = \widetilde{\mathcal{I}}_n. \tag{4.31}$$

The equations to be satisfied by the membrane potentials $\widetilde{\mathcal{V}}_0, \ldots, \widetilde{\mathcal{V}}_n$ are constructed from equations (4.31) by replacing $\widetilde{\mathcal{I}}_L^{(k)}$ and $\widetilde{\mathcal{I}}_R^{(k)}$ with expressions (4.28) and (4.29) respectively. The result is the system of equations

$$\begin{aligned} -\widetilde{\mathcal{V}}_0 \cosh\omega h + \widetilde{\mathcal{V}}_1 &= \frac{\sinh\omega h}{d_1\omega}\widetilde{\mathcal{I}}_0, \\ \frac{d_k}{d_k + d_{k+1}}\widetilde{\mathcal{V}}_{k-1} - \widetilde{\mathcal{V}}_k \cosh\omega h + \frac{d_{k+1}}{d_k + d_{k+1}}\widetilde{\mathcal{V}}_{k+1} &= \frac{\sinh\omega h}{(d_k + d_{k+1})\omega}\widetilde{\mathcal{I}}_k, \\ \widetilde{\mathcal{V}}_{n-1} - \widetilde{\mathcal{V}}_n \cosh\omega h &= \frac{\sinh\omega h}{d_n\omega}\widetilde{\mathcal{I}}_n. \end{aligned} \tag{4.32}$$

The first equation describes the connection of the cable to the parent structure, the second equation describes all internal segments of the cable and the third equation describes the termination of the cable. The tri-diagonal structure of this matrix is apparent. Furthermore, the structure of the equations (4.32) corresponds closely to that described for a branched dendrite in Section 4.2.2. Let $\mathcal{D}$ be the $(n+1) \times (n+1)$ diagonal matrix

$$\mathcal{D} = \text{diag}\,[d_1, (d_1 + d_2), \ldots, (d_k + d_{k+1}), \ldots, (d_{n-1} + d_n), d_n], \tag{4.33}$$

and let $\mathcal{A}$ denote the $(n+1) \times (n+1)$ tri-diagonal matrix with entries

$$\mathcal{A}_{k,k-1} = \frac{d_k}{d_k + d_{k+1}}, \quad \mathcal{A}_{k,k} = -\cosh\omega h, \quad \mathcal{A}_{k,k+1} = \frac{d_{k+1}}{d_k + d_{k+1}}, \tag{4.34}$$

where $d_0 = d_{n+1} = 0$. $\mathcal{A}$ is referred to as the cable matrix. In matrix notation equations (4.32) take the form

$$\mathcal{A}\widetilde{\mathcal{V}} = \frac{\sinh\omega h}{\omega}\mathcal{D}^{-1}\widetilde{\mathcal{I}} \tag{4.35}$$

where

$$\begin{aligned} \widetilde{\mathcal{I}} &= [\widetilde{\mathcal{I}}_0, \widetilde{\mathcal{I}}_1, \ldots, \widetilde{\mathcal{I}}_k, \ldots, \widetilde{\mathcal{I}}_n]^T, \\ \widetilde{\mathcal{V}} &= [\widetilde{\mathcal{V}}_0, \widetilde{\mathcal{V}}_1, \ldots, \widetilde{\mathcal{V}}_k, \ldots, \widetilde{\mathcal{V}}_n]^T \end{aligned} \tag{4.36}$$

are vectors in which the k-th components are respectively the injected current and transmembrane potential at node P_k. Note that the sum of the off-diagonal entries of $\mathcal{A}$ is unity, a feature necessary for the construction of the equivalent cable.

The specification of either injected current or transmembrane potential at each node $P_0, \ldots, P_n$ is necessary to solve equations (4.35). This splits equations (4.35) into two

sets. The first set is for the unknown membrane potentials in terms of the known injected currents, and the second set determines the unknown injected currents from the known membrane potentials.

4.4.3 Symmetrising a cable matrix

The final stage in the development of the canonical form of the piecewise uniform cable is to reduce the tri-diagonal system of equations (4.35) to a symmetric form. Given the cable matrix $\mathcal{A}$ and a non-singular diagonal matrix $\mathcal{S} = \text{diag}\,(1, s_1, \ldots, s_n)$ with inverse $\mathcal{S}^{-1} = \text{diag}\,(1, s_1^{-1}, \ldots, s_n^{-1})$, then there is a choice of $\mathcal{S}$ that will symmetrise $\mathcal{A}$. To see how this is achieved consider the matrix calculation

$$\begin{aligned}
[\mathcal{S}^{-1}(\mathcal{A}\mathcal{S})]_{ij} &= \sum_{r=1}^{n} (\mathcal{S}^{-1})_{ir} (\mathcal{A}\mathcal{S})_{rj} \\
&= \sum_{r=1}^{n} (\mathcal{S}^{-1})_{ir} \sum_{k=1}^{n} \mathcal{A}_{rk} \mathcal{S}_{kj} \\
&= \sum_{r=1}^{n} (\mathcal{S}^{-1})_{ir} \mathcal{A}_{rj} \mathcal{S}_j \\
&= \frac{1}{\mathcal{S}_i} \mathcal{A}_{ij} \mathcal{S}_j.
\end{aligned} \tag{4.37}$$

Since $\mathcal{A}$ is tri-diagonal and $\mathcal{S}$ is diagonal then $T = \mathcal{S}^{-1}\mathcal{A}\mathcal{S}$ is a tri-diagonal matrix with off-diagonal entries

$$T_{k,k+1} = \frac{s_{k+1}}{s_k} \mathcal{A}_{k,k+1}, \qquad T_{k+1,k} = \frac{s_k}{s_{k+1}} \mathcal{A}_{k+1,k}.$$

Symmetry in T requires that $T_{k,k+1} = T_{k+1,k}$ and therefore the elements of $\mathcal{S}$ must be chosen to satisfy

$$\frac{s_{k+1}}{s_k} \mathcal{A}_{k,k+1} = \frac{s_k}{s_{k+1}} \mathcal{A}_{k+1,k}.$$

Thus T will be a symmetric matrix provided

$$s_{k+1} = \pm\, s_k \sqrt{\frac{\mathcal{A}_{k+1,k}}{\mathcal{A}_{k,k+1}}}, \qquad s_0 = 1, \tag{4.38}$$

and in this instance

$$\mathcal{A}_{k+1,k} = \frac{T_{k,k+1}^2}{\mathcal{A}_{k,k+1}} \quad \rightarrow \quad T_{k,k+1} = T_{k+1,k} = \sqrt{\mathcal{A}_{k+1,k}\mathcal{A}_{k,k+1}}. \tag{4.39}$$

Since every off-diagonal entry of the cable matrix $\mathcal{A}$ is positive, then $\mathcal{S}$ is a real matrix. However, one must select the appropriate algebraic sign in equation (4.38) to ensure that

$\mathcal{A}_{k,k+1}$ is positive. Once $\mathcal{S}$ is constructed, equation (4.35) can now be expressed in the symmetric tri-diagonal form by noting that the original equation

$$\mathcal{A}\widetilde{\mathcal{V}} = \frac{\sinh \omega h}{\omega} \mathcal{D}^{-1}\widetilde{\mathcal{I}}$$

can be expressed in the form

$$\mathcal{S}^{-1}\mathcal{A}\mathcal{S}\mathcal{S}^{-1}\widetilde{\mathcal{V}} = \frac{\sinh \omega h}{\omega} \mathcal{S}^{-1}\mathcal{D}^{-1}\widetilde{\mathcal{I}}$$

which in turn shows that the canonical representation of a piecewise uniform cable is

$$T(\mathcal{S}^{-1}\widetilde{\mathcal{V}}) = \frac{\sinh \omega h}{\omega} (\mathcal{D}\mathcal{S})^{-1}\widetilde{\mathcal{I}}. \tag{4.40}$$

The aim of the following sections is to demonstrate that the mathematical representation of a branched dendrite can be reduced to this form.

4.4.4 Structure of tree matrices

The model equations for the construction of a branched dendrite with $(n+1)$ nodes (see Section 4.2.2) leads to the matrix representation

$$A\widetilde{V} = \frac{\sinh \omega h}{\omega} D^{-1} \widetilde{I} \tag{4.41}$$

where $\widetilde{V}$ and $\widetilde{I}$ are respectively the vectors of the Laplace transforms of membrane potentials and injected currents. The $(n+1) \times (n+1)$ matrix A is referred to as the tree matrix and it is neither tri-diagonal nor symmetric.

The construction of equivalent cables from dendritic structures depends critically on the fact that any dendritic structure characterised by $(n+1)$ nodes has a tree matrix A consisting of $(n+1)$ non-zero diagonal entries, one for each node, and $2n$ positive off-diagonal entries distributed symmetrically about the main diagonal of A, giving a total of $3n+1$ non-zero entries. The matrix A is structured symmetrically from the observation that if node j is connected to node k then node k is connected to node j. The number of non-zero off-diagonal elements of A is established by taking advantage of the self-similarity inherent in a branched structure by using a recursive counting argument.

Self-similarity argument

In a dendritic tree, a node can be classed as one of four types: the first node which connects with the parent structure, an internal node, a branch point node or a dendritic terminal. The self-similarity argument starts with a terminal node and is applied recursively until the node connected to the parent structure is reached. The process involves counting the deficit in off-diagonal entries in A with respect to two entries per node. Internal nodes have no deficit while each dendritic terminal has a deficit of one. If N terminal segments and a parent segment meet at a branch point node, then the row of A corresponding to that branch point contains $(N+1)$ off-diagonal entries giving a surplus of $(N-1)$. Therefore the total deficit is reduced to one at the branch point node. This node then behaves like a dendritic terminal with respect to further counting. The deficit of one is maintained until the node connecting to the parent structure is reached, at which point the deficit increases to two nodes for the entire tree. Therefore A contains exactly n pairs of non-zero off-diagonal entries.

Node numbering

As already mentioned, a distinction is made between nodes at which membrane potential or injected current has to be specified. This distinction also applies for the process of node numbering. Nodes at which the injected current is known and the potential has to be found are numbered first and then the numbering moves to those nodes where membrane potential is known and injected current has to be found. Figure 4.3 shows the node numbering method for a dendrite where the injected current is known at *all* nodes (4.3A) and a dendrite where the potential is known at two nodes (4.3B).

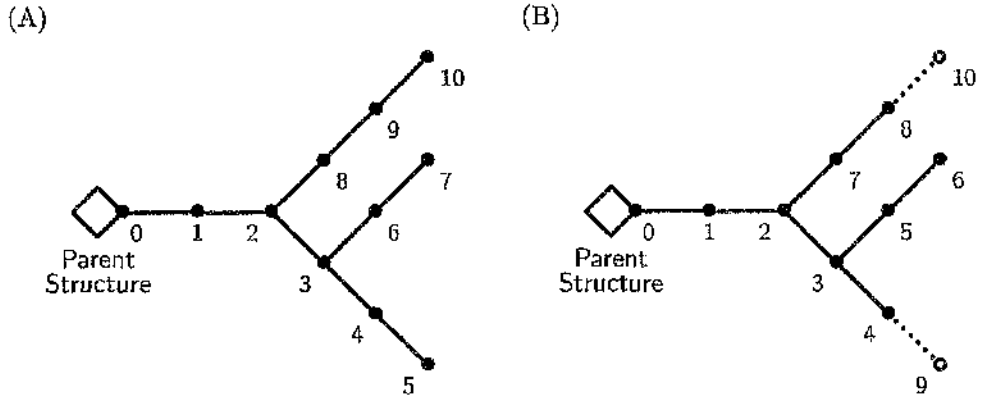

Figure 4.3: The enumeration scheme for (A), a dendrite where the injected current is known at all nodes and the membrane potential has to be determined and (B), the same dendrite however the membrane potential is specified at two dendritic terminals (nodes 9 and 10).

The enumeration scheme starts at P_0, the connection to the parent structure. From the parent structure, the nodes are numbered sequentially until reaching a dendritic terminal, omitting nodes where the potential is known. The numbering then jumps to a second path, starting with a node that has a connection to the first path and again continues until reaching a dendritic terminal, omitting nodes with known potential. The enumeration scheme is repeated until all dendritic paths have been numbered and then repeats numbering those nodes at which the potential is known.

Let the entries of tree matrix A be represented by black squares, then the matrix repre-

sentation of the dendrite in Figure 4.3A is

$$\begin{bmatrix} \blacksquare & \blacksquare & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \blacksquare & \blacksquare & \blacksquare & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \blacksquare & \blacksquare & \blacksquare & 0 & 0 & 0 & 0 & \blacksquare & 0 & 0 \\ 0 & 0 & \blacksquare & \blacksquare & \blacksquare & 0 & \blacksquare & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \blacksquare & \blacksquare & \blacksquare & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \blacksquare & \blacksquare & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \blacksquare & 0 & 0 & \blacksquare & \blacksquare & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \blacksquare & \blacksquare & 0 & 0 & 0 \\ 0 & 0 & \blacksquare & 0 & 0 & 0 & 0 & 0 & \blacksquare & \blacksquare & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \blacksquare & \blacksquare & \blacksquare \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \blacksquare & \blacksquare \end{bmatrix} \begin{bmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ V_6 \\ V_7 \\ V_8 \\ V_9 \\ V_{10} \end{bmatrix} = \frac{\sinh \omega h}{\omega} \begin{bmatrix} I_0/D_{0,0} \\ I_1/D_{1,1} \\ I_2/D_{2,2} \\ I_3/D_{3,3} \\ I_4/D_{4,4} \\ I_5/D_{5,5} \\ I_6/D_{6,6} \\ I_7/D_{7,7} \\ I_8/D_{8,8} \\ I_9/D_{9,9} \\ I_{10}/D_{10,10} \end{bmatrix}.$$

The node numbering scheme for the second dendrite (Figure 4.3B), where the membrane potential is specified at nodes 9 and 10 has matrix representation

$$\left[\begin{array}{ccccccccc|cc} \blacksquare & \blacksquare & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \blacksquare & \blacksquare & \blacksquare & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \blacksquare & \blacksquare & \blacksquare & 0 & 0 & 0 & \blacksquare & 0 & 0 & 0 \\ 0 & 0 & \blacksquare & \blacksquare & \blacksquare & \blacksquare & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \blacksquare & \blacksquare & 0 & 0 & 0 & 0 & \blacksquare & 0 \\ 0 & 0 & 0 & \blacksquare & 0 & \blacksquare & \blacksquare & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \blacksquare & \blacksquare & 0 & 0 & 0 & 0 \\ 0 & 0 & \blacksquare & 0 & 0 & 0 & 0 & \blacksquare & \blacksquare & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \blacksquare & \blacksquare & 0 & \blacksquare \\ \hline 0 & 0 & 0 & 0 & \blacksquare & 0 & 0 & 0 & 0 & \blacksquare & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \blacksquare & 0 & \blacksquare \end{array}\right] \left[\begin{array}{c} V_0 \\ V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ V_6 \\ V_7 \\ V_8 \\ \hline V_9 \\ V_{10} \end{array}\right] = \frac{\sinh \omega h}{\omega} \left[\begin{array}{c} I_0/D_{0,0} \\ I_1/D_{1,1} \\ I_2/D_{2,2} \\ I_3/D_{3,3} \\ I_4/D_{4,4} \\ I_5/D_{5,5} \\ I_6/D_{6,6} \\ I_7/D_{7,7} \\ I_8/D_{8,8} \\ \hline I_9/D_{9,9} \\ I_{10}/D_{10,10} \end{array}\right],$$

where the non-zero elements are denoted by black squares. The matrix is partitioned such that the 9×9 block represents the nodes at which the injected current is known and the membrane potential has to be determined. The remaining partitions represent the nodes where the injected current has to be determined from the membrane potential specified at nodes 9 and 10.

4.4.5 Symmetrising the tree matrix

It was shown in Section 4.4.4 that a tree matrix with dimension $(n+1) \times (n+1)$ has $(n+1)$ diagonal entries and $2n$ positive off-diagonal entries, where $A_{k,j} \neq 0$ if and only if $A_{j,k} \neq 0$. Furthermore, in Section 4.4.3, it was shown that given any cable matrix $\mathcal{A}$ it is possible to find a non-singular diagonal matrix S such that $S^{-1}\mathcal{A}S$ is a symmetric matrix. Tree matrices can be symmetrised in the same fashion. Let $S = \text{diag}\,(\mathbf{1}, s_1, \ldots, s_n)$ be a non-singular $(n+1) \times (n+1)$ diagonal matrix, then $S^{-1}AS$ is the $(n+1) \times (n+1)$ matrix with entries

$$[S^{-1}AS]_{j,k} = \frac{A_{j,k}\, s_k}{s_j}, \qquad [S^{-1}AS]_{k,j} = \frac{A_{k,j}\, s_j}{s_k}. \tag{4.42}$$

Provided that there is a matrix S, such that the entries of $S^{-1}AS$ satisfy $[S^{-1}AS]_{j,k} = [S^{-1}AS]_{k,j}$ for all $j \neq k$ and $A_{j,k} \neq 0$, $A_{k,j} \neq 0$, then the matrix $S^{-1}AS$ will be symmetric. From Section 4.4.4, A has n non-zero pairs of entries for which $A_{j,k} \neq 0$, $A_{k,j} \neq 0$ where $j \neq k$. Each pair of equations contributes an equation of the form

$$s_k = s_j \sqrt{\frac{A_{k,j}}{A_{j,k}}}, \tag{4.43}$$

giving in total a system of n equations to determine the n unknowns $s_1, \ldots, s_n$. The corresponding symmetrised tree matrix is

$$[S^{-1}AS]_{j,k} = [S^{-1}AS]_{k,j} = \sqrt{A_{j,k}A_{k,j}}. \tag{4.44}$$

Equation (4.43) determines all the entries of S, from the observation that there are n such equations and that each node on the tree is connected to *at least* one other node. Therefore, there is no entry of S which does not appear in at least one of equations (4.43). As $s_0 = 1$ is the initial condition of equation (4.43), then $s_1, \ldots, s_n$ will be determined uniquely meaning that every tree matrix can be symmetrised by an appropriate choice of non-singular diagonal matrix $S = \text{diag}\,(1, s_1, \ldots, s_n)$. Pre-multiplication of the system of equations for a branched model dendrite (4.41) by S^{-1} gives the symmetric form

$$(S^{-1}AS)S^{-1}\widetilde{V} = \frac{\sinh \omega h}{\omega} (DS)^{-1}\widetilde{I}. \tag{4.45}$$

4.4.6 Tri-diagonalising the symmetric tree matrix

The remaining task is to reduce the symmetrised tree matrix (4.45) into the canonical form for a piecewise uniform cable (4.40). The symmetric tree matrix is reduced to a tri-diagonal symmetric matrix by applying a series of Householder transformations (Lindsay,

Ogden, Halliday and Rosenberg, 1999; Golub and Van Loan, 1989; also see Section 4.8.1). The resulting tri-diagonal symmetric matrix can then be interpreted as a symmetrised cable matrix.

If H is the symmetrising Householder matrix, then $T = (SH)^{-1}A(SH)$ is a tri-diagonal symmetric matrix. Pre-multiplication of equation (4.45) by H^{-1} reduces the symmetric matrix to a tri-diagonal symmetric matrix through the series of matrix manipulations

$$\begin{aligned}
(S^{-1}AS)S^{-1}\tilde{V} &= \frac{\sinh\omega h}{\omega}(DS)^{-1}\tilde{I}, \\
H^{-1}(S^{-1}AS)H\,H^{-1}S^{-1}\tilde{V} &= \frac{\sinh\omega h}{\omega}H^{-1}(DS)^{-1}\tilde{I}, \\
(SH)^{-1}A(SH)\,(SH)^{-1}\tilde{V} &= \frac{\sinh\omega h}{\omega}(DSH)^{-1}\tilde{I}, \\
T(SH)^{-1}\tilde{V} &= \frac{\sinh\omega h}{\omega}(DSH)^{-1}\tilde{I}.
\end{aligned} \tag{4.46}$$

If T is interpreted as the symmetric form of a cable matrix, both $\mathcal{S}$ and the equivalent cable matrix $\mathcal{A}$ may be obtained.

4.4.7 Mapping of potentials and currents from tree to equivalent cable

A direct comparison of the equations representing the symmetrised piecewise uniform cable and those representing the tri-diagonalised symmetric branched dendrite leads to a mapping of potentials and currents from the branched to unbranched structure. The system equations for the (symmetrised) piecewise uniform cable and those for the tri-diagonalised symmetric branched dendrite from Sections 4.4.3 and 4.4.6 respectively are

$$\begin{aligned}
T(\mathcal{S}^{-1}\tilde{\mathcal{V}}) &= \frac{\sinh\omega h}{\omega}(\mathcal{D}\mathcal{S})^{-1}\tilde{\mathcal{I}}, \\
T(SH)^{-1}\tilde{V} &= \frac{\sinh\omega h}{\omega}(DSH)^{-1}\tilde{I}.
\end{aligned} \tag{4.47}$$

Equations (4.47) are identical provided membrane potentials and injected currents on the unbranched and branched dendrites are connected by the formulae

$$\mathcal{S}^{-1}\tilde{\mathcal{V}} = (SH)^{-1}\tilde{V}, \qquad (\mathcal{D}\mathcal{S})^{-1}\tilde{\mathcal{I}} = (DSH)^{-1}\tilde{I}. \tag{4.48}$$

Formulae (4.48) relate potentials and injected currents on the branched dendrite to those on the equivalent cable. By inverting the Laplace transforms in equations (4.48), the potentials and injected currents on the tree are related to those on the cable by the

formulae

$$\begin{aligned} S^{-1}\mathcal{V}(s) &= (SH)^{-1}V(s), \quad \rightarrow \quad \mathcal{V}(s) = \Psi_V V(s) \\ (\mathcal{D}\mathcal{S})^{-1}\mathcal{I}(s) &= (DSH)^{-1}I(s) \quad \rightarrow \quad \mathcal{I}(s) = \Psi_C I(s). \end{aligned} \tag{4.49}$$

The matrices Ψ_V and Ψ_C appearing in equations (4.49) are called the voltage and current electro-geometric projection (EGP) matrices respectively, and are defined by

$$\Psi_V = \mathcal{S}H^T S^{-1}, \qquad \Psi_C = \mathcal{D}(\mathcal{S}H^T S^{-1})D^{-1} = \mathcal{D}\Psi_V D^{-1}. \tag{4.50}$$

The EGP matrices are by-products of the construction procedure. They are determined by the characteristic conductances of the branched dendrite, and therefore its biophysical and geometrical properties. To construct the equivalent cable it remains to calculate the symmetrising matrix $\mathcal{S}$, the equivalent cable matrix $\mathcal{A}$ from T, and the characteristic conductance of each cable section.

4.4.8 Construction of the equivalent cable

Constructing the unbranched cable proceeds in two steps. First, $\mathcal{A}$ is constructed and this in turn provides the characteristic conductances and dimensions of the equivalent cable. Second, $\mathcal{D}$ is constructed which is then used to determine the one-to-one mapping of injected currents on the branched model to those on the equivalent cable.

Characteristic conductances and equivalent cable dimensions

T is interpreted as a tri-diagonal symmetrised cable matrix from which $\mathcal{S}$ and $\mathcal{A}$ are to be determined. The extraction of the equivalent cable from T uses the fact that

$$\mathcal{A}_{k,k-1} + \mathcal{A}_{k,k+1} = 1. \tag{4.51}$$

That is, the sum of the off-diagonal entries in each row is unity, for suitable values of k. Furthermore, a cable matrix $\mathcal{A}$ corresponding to a piecewise uniform cable with n sections has dimension $(n+1) \times (n+1)$ and satisfies $\mathcal{A}_{0,1} = \mathcal{A}_{n,n-1} = 1$. This relation in combination with (4.51) allows the extraction of $\mathcal{A}$ using the expression

$$\mathcal{A}_{k+1,k} = \frac{T^2_{k,k+1}}{\mathcal{A}_{k,k+1}}, \tag{4.52}$$

which subsequently allows the extraction of $\mathcal{S}$ from $\mathcal{A}$ using

$$s_{k+1} = \pm s_k \sqrt{\frac{\mathcal{A}_{k+1,k}}{\mathcal{A}_{k,k+1}}}, \qquad s_0 = 1. \tag{4.53}$$

This is the analytical method of extracting $\mathcal{A}$ and $\mathcal{S}$ from the matrix T. However, in numerical work, in order to avoid rounding error in the repeated calculation of $\mathcal{A}_{k,k+1}$ from $\mathcal{A}_{k,k-1}$ via the formula $\mathcal{A}_{k,k+1} = 1 - \mathcal{A}_{k,k-1}$ it is beneficial to satisfy this condition identically by the representation

$$\mathcal{A}_{k,k-1} = \cos^2 \theta_k, \qquad \mathcal{A}_{k,k+1} = \sin^2 \theta_k. \tag{4.54}$$

With this representation of the entries of the cable matrix, the iteration procedure begins with $\theta_0 = \pi/2$ ($\mathcal{A}_{0,1} = 1$) and ends when $\theta_n = 0$ ($\mathcal{A}_{n,n-1} = 1$). Of course, in a numerical calculation the cable section will be deemed to be complete when $\theta_n < \epsilon$ where ϵ is a user-supplied small number. The entries of the cable matrix are constructed from the condition (4.52), expressed in the iterative form

$$\theta_{k+1} = \cos^{-1}\left(\frac{|T_{k,k+1}|}{\sin\theta_k}\right), \qquad \theta_0 = \frac{\pi}{2}. \tag{4.55}$$

The characteristic conductances of the individual cable sections are determined from the definitions (4.34) of $\mathcal{A}_{k,k-1}$ and $\mathcal{A}_{k,k+1}$ by the iterative formula

$$d_{k+1} = d_k \tan^2 \theta_k, \qquad d_1 \text{ given.} \tag{4.56}$$

This allows the characteristic conductances of each section to be calculated without directly calculating $\mathcal{A}$ and $\mathcal{S}$. Given the characteristic conductances, the section diameters of the equivalent cable are found from expression (4.6) assuming a piecewise uniform cable.

Determination of real input currents on the equivalent cable

The construction of the vector of real input currents, $\mathcal{I}$, on the cable requires the computation of Ψ_C in equation (4.50). The Householder matrix H, the symmetrising tree matrix S and the diagonalising tree matrix D arise in the construction of the tree matrix and are therefore known. However, the symmetrising cable matrix $\mathcal{S}$ and the diagonal cable matrix $\mathcal{D}$ still need to be determined. The symmetrising cable matrix $\mathcal{S}$ can be calculated from the formula

$$\mathcal{S}_{k+1} = \pm\, \mathcal{S}_k \sqrt{\frac{\mathcal{A}_{k+1,k}}{\mathcal{A}_{k,k+1}}} \tag{4.57}$$

where

$$\mathcal{A}_{k,\,k+1} = \frac{d_{k+1}}{d_k + d_{k+1}} \quad \text{and} \quad \mathcal{A}_{k+1,\,k} = \frac{d_{k+1}}{d_{k+1} + d_{k+2}}. \tag{4.58}$$

The algebraic sign in (4.57) is chosen to ensure that all values in the final cable matrix $\mathcal{A}$ are positive. This decision is made during the construction of $\mathcal{S}$ as subsequent entries

of S depend on previous entries. The characteristic conductances for each cable section are calculated analytically from (4.58) and numerically from (4.56). The final form of $\mathcal{D}$ is formed from the definition

$$\mathcal{D} = \text{diag } [d_1, (d_1 + d_2), \ldots, (d_k + d_{k+1}), \ldots, (d_{n-1} + d_n), d_n]. \tag{4.59}$$

From here it is straightforward matrix multiplication to find Ψ_C and subsequently $\mathcal{I}(s)$.

4.4.9 Summary - Concept of the equivalent cable

In summarising the work of the previous sections, the concept of an equivalent cable follows from the observation that under certain conditions a symmetric tri-diagonal matrix may be interpreted as a cable matrix. The equivalent cable is the result of a series of transformations that convert a tree matrix into a symmetric tri-diagonal matrix. This matrix in turn may be interpreted as a symmetrised cable matrix. The associated piecewise uniform cable represented by the latter is defined as the equivalent cable of the branched dendrite. In addition, the construction process provides a procedure by which the distribution of inputs on the branched structure can be uniquely mapped to those on the equivalent cable and conversely.

4.5 Analytical construction of an equivalent cable

The manual construction of an equivalent cable is feasible only for simple branched structures, but of course, the analytical procedure can be implemented numerically for cables of arbitrary size. This section describes the exact construction of the equivalent cable for three examples with increasing geometrical complexity. These examples highlight all the features of the equivalent cable including, the extraction of the equivalent cable from the cable matrix and the mapping of potentials and currents on the branched model dendrite to the unbranched model. In addition, the distinction between the equivalent cable and Rall's equivalent cylinder will be made precise.

4.5.1 A simple Rall branch point

Figure 4.4 shows a simple Y-junction with two limbs of electrotonic length h meeting at P_0. Exogenous currents I_1 and I_2 are injected at points P_1 and P_2, while core current I_0 flows from the Y-junction to its parent structure at P_0. This Y-junction immediately satisfies the Rall condition that both segments forming the Y-junction have equal electrotonic length.

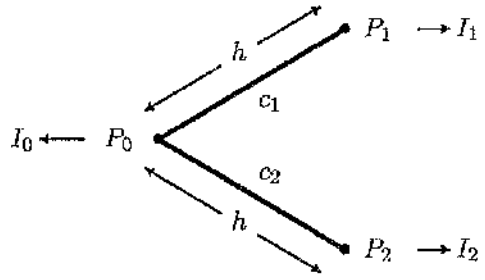

Figure 4.4: A symmetric Y-junction with limbs of equal electrotonic length. The sections joining P_0 to P_1 and P_0 to P_2 have length h and characteristic conductances c_1 and c_2 respectively.

Particularisation of equations (4.28) and (4.29) to the junction in Figure 4.4, shows that the Laplace transforms of the membrane potentials $V_0(s)$, $V_1(s)$ and $V_2(s)$ at points P_0,

P_1 and P_2 satisfy the algebraic equations

$$\begin{aligned} -\widetilde{V}_0 \cosh \omega h + \frac{c_1}{c_1+c_2}\widetilde{V}_1 + \frac{c_2}{c_1+c_2}\widetilde{V}_2 &= \frac{\sinh \omega h}{(c_1+c_2)\,\omega}\,\widetilde{I}_0\,, \\ \widetilde{V}_0 - \widetilde{V}_1 \cosh \omega h &= \frac{\sinh \omega h}{c_1\,\omega}\,\widetilde{I}_1\,, \\ \widetilde{V}_0 - \widetilde{V}_2 \cosh \omega h &= \frac{\sinh \omega h}{c_2\,\omega}\,\widetilde{I}_2\,. \end{aligned} \tag{4.60}$$

These equations have matrix representation,

$$A\widetilde{V} = \frac{\sinh \omega h}{\omega}\, D^{-1}\widetilde{I} \tag{4.61}$$

where $\widetilde{V}$ is the column vector of the Laplace transforms of the membrane potentials at points P_0, P_1, P_2 and $\widetilde{I}$ is the corresponding column vector of Laplace transforms of the injected currents at these points. The tree matrix is

$$A = \begin{bmatrix} -\cosh \omega h & \dfrac{c_1}{c_1+c_2} & \dfrac{c_2}{c_1+c_2} \\ 1 & -\cosh \omega h & 0 \\ 1 & 0 & -\cosh \omega h. \end{bmatrix} \tag{4.62}$$

and D is the diagonal matrix with k-th entry equal to the sum of the characteristic conductances of the sections which meet at the k-th node. Thus

$$D = \text{ diag } \left[c_1 + c_2,\, c_1,\, c_2\right]. \tag{4.63}$$

Following the procedure described in Section 4.4.5, in particular expressions (4.43) and (4.44), it can be shown that the diagonal matrix

$$S = \text{diag} \left[1, \sqrt{\frac{c_1+c_2}{c_1}}, \sqrt{\frac{c_1+c_2}{c_2}}\right]$$

reduces the tree matrix A to the symmetric form

$$\begin{bmatrix} -\cosh \omega h & \sqrt{\dfrac{c_1}{c_1+c_2}} & \sqrt{\dfrac{c_2}{c_1+c_2}} \\ \sqrt{\dfrac{c_1}{c_1+c_2}} & -\cosh \omega h & 0 \\ \sqrt{\dfrac{c_2}{c_1+c_2}} & 0 & -\cosh \omega h \end{bmatrix}, \tag{4.64}$$

which leads to the system representation

$$(S^{-1}AS)\, S^{-1}\widetilde{V} = \frac{\sinh \omega h}{\omega}\, (DS)^{-1}\, \widetilde{I}. \tag{4.65}$$

The next stage of the construction procedure requires $S^{-1}AS$ to be transformed into a tri-diagonal symmetric matrix. This is achieved using the orthogonal symmetric matrix

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \sqrt{\dfrac{c_1}{c_1+c_2}} & \sqrt{\dfrac{c_2}{c_1+c_2}} \\ 0 & \sqrt{\dfrac{c_2}{c_1+c_2}} & -\sqrt{\dfrac{c_1}{c_1+c_2}} \end{bmatrix}. \tag{4.66}$$

The matrix H is derived in a systematic way as a product of a finite sequence of Householder transformations described in Section 4.8.1 (see also Lindsay *et al.*, 1999; Golub and Van Loan, 1989). Since T is a tri-diagonal symmetric matrix, it may be interpreted as the symmetric form of a cable matrix. The construction process is completed by pre-multiplying equation (4.65) by H^{-1} to obtain

$$T(SH)^{-1}\widetilde{V} = \frac{\sinh\omega h}{\omega}(DSH)^{-1}\widetilde{I} \tag{4.67}$$

where $T = (SH)^{-1}\Lambda(SH)$. The resulting system of equations takes the form

$$\begin{bmatrix} -\cosh\omega h & 1 & 0 \\ 1 & -\cosh\omega h & 0 \\ 0 & 0 & \cosh\omega h \end{bmatrix} \begin{bmatrix} \widetilde{V}_0 \\ \dfrac{c_1\widetilde{V}_1 + c_2\widetilde{V}_2}{c_1+c_2} \\ \dfrac{\sqrt{c_1c_2}(\widetilde{V}_1 - \widetilde{V}_2)}{c_1+c_2} \end{bmatrix} = \frac{\sinh\omega h}{\omega} \begin{bmatrix} \dfrac{\widetilde{I}_0}{c_1+c_2} \\ \dfrac{\widetilde{I}_1 + \widetilde{I}_2}{c_1+c_2} \\ \dfrac{c_2\widetilde{I}_1 - c_1\widetilde{I}_2}{\sqrt{c_1c_2}\,(c_1+c_2)} \end{bmatrix}. \tag{4.68}$$

Equation (4.68) is the tri-diagonal symmetrised form of the original Y-junction. The Y-junction can be represented as a cable by showing that equation (4.68) can be associated with a cable matrix. The characteristic conductances of the cable sections and the mapping between injected currents on the Y-junction and those on the cable can then be extracted.

Equations (4.68) divide naturally into the 2×2 system

$$\begin{bmatrix} -\cosh\omega h & 1 \\ 1 & -\cosh\omega h \end{bmatrix} \begin{bmatrix} \widetilde{V}_0 \\ \dfrac{c_1\widetilde{V}_1 + c_2\widetilde{V}_2}{c_1+c_2} \end{bmatrix} = \frac{\sinh\omega h}{\omega} \begin{bmatrix} \dfrac{\widetilde{I}_0}{c_1+c_2} \\ \dfrac{\widetilde{I}_1 + \widetilde{I}_2}{c_1+c_2} \end{bmatrix} \tag{4.69}$$

and the single equation

$$-\frac{\sqrt{c_1c_2}(\widetilde{V}_1 - \widetilde{V}_2)}{c_1+c_2}\cosh\omega h = \frac{\sinh\omega h}{\omega}\left[\frac{c_2\widetilde{I}_1 - c_1\widetilde{I}_2}{\sqrt{c_1c_2}\,(c_1+c_2)}\right]. \tag{4.70}$$

Equations (4.69) and (4.70) can be interpreted as two equivalent cables, one of which is connected to the parent structure at the branch point in Figure 4.4 and will be called the *connected cable*, and the other which will be shown to be disconnected from the parent structure and will be called a *disconnected cable*.

Connected cable

The extraction of the connected section of the equivalent cable is achieved by comparing equations (4.69) with the known form for a cable of length h, namely

$$\begin{bmatrix} -\cosh\omega h & 1 \\ 1 & -\cosh\omega h \end{bmatrix} \begin{bmatrix} \widetilde{\mathcal{V}}_0 \\ \widetilde{\mathcal{V}}_1 \end{bmatrix} = \frac{\sinh\omega h}{\omega} \begin{bmatrix} \dfrac{\widetilde{\mathcal{I}}_0}{d_1} \\ \dfrac{\widetilde{\mathcal{I}}_1}{d_2} \end{bmatrix}. \tag{4.71}$$

The identifications

$$\begin{aligned} \mathcal{V}_0 &= V_0, & d_1 &= c_1 + c_2, \\ \mathcal{I}_0 &= I_0, & \mathcal{I}_1 &= I_1 + I_2 \end{aligned} \tag{4.72}$$

render equation (4.69) structurally identical to (4.71). The left-hand pair of identities in (4.72) guarantees continuity of membrane potential and conservation of core current at the point of connection of the Y-junction to the parent structure. The right-hand pair in (4.72) determines the characteristic conductance of the first section of the equivalent cable and the injected current at its distal end.

Disconnected cable

The single equation (4.70) is now compared with the first equation in the general representation of a cable (4.71) of length h, namely

$$-\cosh\omega h\widetilde{\mathcal{V}}_0 + \widetilde{\mathcal{V}}_1 = \frac{\sinh\omega h}{d_1\omega}\widetilde{\mathcal{I}}_0. \tag{4.73}$$

These equations are identical provided

$$\begin{aligned} \mathcal{V}_0 &= \frac{\sqrt{c_1c_2}\,(V_1 - V_2)}{c_1 + c_2}, & d_1 &= c_1 + c_2, \\ \mathcal{I}_0 &= \sqrt{\frac{c_2}{c_1}}\,I_1 - \sqrt{\frac{c_1}{c_2}}\,I_2, & \mathcal{V}_1 &= 0, \end{aligned} \tag{4.74}$$

where the value of d_1 is arbitrary and in this case is chosen to be $c_1 + c_2$. Under these circumstances the second equation of (4.71) now specifies the current to be injected at the

distal end of the cable to maintain zero potential. However, unlike the connected cable, this cable is not unique. For example, expression (4.70) may be rewritten in the equivalent mathematical form

$$-(\widetilde{V}_1 - \widetilde{V}_2)\cosh\omega h = \frac{\sinh\omega h}{\omega c_2}\left[\frac{c_2}{c_1}\widetilde{I}_1 - \widetilde{I}_2\right]. \tag{4.75}$$

Direct comparison of this equation with the first equation in the general representation (4.71) leads to the identities

$$\begin{aligned} \mathcal{V}_0 &= V_1 - V_2 \qquad & d_1 &= c_2, \\ \mathcal{I}_0 &= \frac{c_2}{c_1} I_1 - I_2, \qquad & \mathcal{V}_1 &= 0. \end{aligned} \tag{4.76}$$

Equation (4.76) is obtained by re-scaling equation (4.74). This means that the characteristic conductance of the second cable is arbitrary, but once given a value, the membrane potentials and injected currents on the second section are determined uniquely. The non-uniqueness of the disconnected cable does not affect the properties of the connected cable since the former is isolated electrically from the latter, and therefore from the parent structure.

Summary of equivalent cable

It has been shown that a Y-junction with limbs of equal electrotonic length h and current injected tips has an equivalent cable with electrotonic length $2h$ that is composed of two independent cables of electrotonic length h, only one of which is connected to the parent structure. If c_1 and c_2 are respectively the characteristic conductances of the limbs of the Y-junction and I_1 and I_2 are the currents injected at its terminals, then the equivalent cable of the Y-junction has conductances and current mappings

$$\begin{aligned} &\text{Connected section} \quad \begin{cases} d_1 = c_1 + c_2 \\ \mathcal{I}_1 = I_1 + I_2, \end{cases} \\ &\text{Disconnected section} \quad \begin{cases} d_2 = c_1 + c_2 \\ \mathcal{I}_2 = (c_1 + c_2)\left[\dfrac{I_1}{c_1} - \dfrac{I_2}{c_2}\right]. \end{cases} \end{aligned} \tag{4.77}$$

This cable is equivalent to the original Y-junction because it preserves continuity of membrane potential and conservation of core current at the point of connection with the parent

structure, and additionally, any configuration of injected currents on the Y-junction defines a unique configuration of injected currents on the cable, and *vice versa*.

The distinction between the equivalent cable and Rall's equivalent cylinder becomes clear from this example. A cable of electrotonic length $2h$ has been constructed. The connected section of this cable with length h is Rall's equivalent cylinder. The role of the disconnected section of this cable, also of length h, is to complete the one-to-one mapping between input on the Y-junction and that on the equivalent cable. Evidently, Rall's equivalent cylinder is exact but deficient in the respect that there is no one-to-one mapping between the currents on the tree and those on the cylinder.

4.5.2 An asymmetric Y-junction

Figure 4.5 shows an asymmetric Y-junction with limbs of (unequal) length $2h$ and h meeting at the parent structure P_0. Currents I_1, I_2 and I_3 are injected at P_1, P_2 and P_3 respectively, and core current I_0 flows from the Y-junction to its parent structure at P_0. Clearly Rall's equivalent cylinder could not be constructed for this Y-junction since the electrotonic length of the limbs of the Y-junction from the point of connection to the parent structure to each terminal is different.

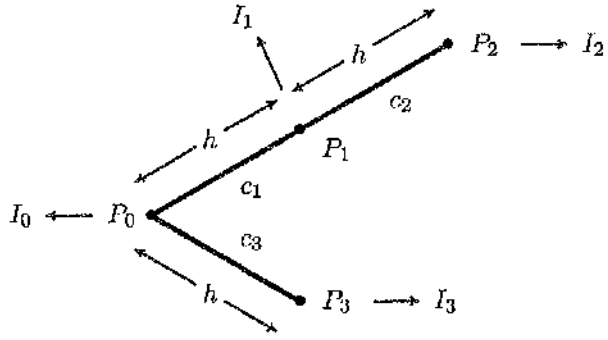

Figure 4.5: A Y-junction with limbs of electrotonic length $2h$ and h. The sections joining P_0 to P_1, P_1 to P_2 and P_0 to P_3 each have electrotonic length h and characteristic conductances c_1, c_2 and c_3 respectively.

The Laplace transforms of the membrane potentials $V_0, \ldots, V_3$ at the $P_0, \ldots, P_3$ respectively, when equations (4.28) and (4.29) have been particularised to the asymmetric Y-

junction in Figure 4.5, satisfy the algebraic equations

$$\begin{aligned}
-\widetilde{V}_0 \cosh\omega h + \frac{c_1}{c_1+c_3}\widetilde{V}_1 + \frac{c_3}{c_1+c_3}\widetilde{V}_3 &= \frac{\sinh\omega h}{\omega(c_1+c_3)}\,\widetilde{I}_0, \\
\frac{c_1}{c_1+c_2}\widetilde{V}_0 - \widetilde{V}_1\cosh\omega h + \frac{c_2}{c_1+c_2}\widetilde{V}_2 &= \frac{\sinh\omega h}{\omega(c_1+c_2)}\,\widetilde{I}_1, \\
\widetilde{V}_1 - \widetilde{V}_2\cosh\omega h &= \frac{\sinh\omega h}{\omega c_2}\,\widetilde{I}_2, \\
\widetilde{V}_0 - \widetilde{V}_3\cosh\omega h &= \frac{\sinh\omega h}{\omega c_3}\,\widetilde{I}_3.
\end{aligned} \tag{4.78}$$

These equations have matrix representation

$$A\widetilde{V} = \frac{\sinh\omega h}{\omega}\,D^{-1}\,\widetilde{I} \tag{4.79}$$

where $\widetilde{V}$ is the column vector of the Laplace transforms of the membrane potentials at points $P_0, \ldots, P_3$, $\widetilde{I}$ is the corresponding column vector of Laplace transforms of the injected currents at these points, and A is the tree matrix

$$\begin{bmatrix}
-\cosh\omega h & \dfrac{c_1}{c_1+c_3} & 0 & \dfrac{c_3}{c_1+c_3} \\
\dfrac{c_1}{c_1+c_2} & -\cosh\omega h & \dfrac{c_2}{c_1+c_2} & 0 \\
0 & 1 & -\cosh\omega h & 0 \\
1 & 0 & 0 & -\cosh\omega h
\end{bmatrix}. \tag{4.80}$$

The k-th entry of the diagonal matrix D is the sum of the characteristic conductances of the sections which meet at the k-th node. In this example,

$$D = \operatorname{diag}\left[c_1+c_3,\ c_1+c_2,\ c_2,\ c_3\right]. \tag{4.81}$$

Following the procedure described in Section 4.4.5, it can be shown that the diagonal matrix

$$S = \operatorname{diag}\left[1,\ \sqrt{\frac{c_1+c_3}{c_1+c_2}},\ \sqrt{\frac{c_1+c_3}{c_2}},\ \sqrt{\frac{c_1+c_3}{c_3}}\right] \tag{4.82}$$

will symmetrise the tree matrix A to obtain

$$S^{-1}AS = \begin{bmatrix}
-\cosh\omega h & p & 0 & q \\
p & -\cosh\omega h & r & 0 \\
0 & r & -\cosh\omega h & 0 \\
q & 0 & 0 & -\cosh\omega h
\end{bmatrix} \tag{4.83}$$

where p, q and r represent the expressions

$$p = \frac{c_1}{\sqrt{(c_1+c_2)(c_1+c_3)}}, \qquad q = \sqrt{\frac{c_3}{c_1+c_3}}, \qquad r = \sqrt{\frac{c_2}{c_1+c_2}}. \tag{4.84}$$

The first step in the construction of the equivalent cable is to pre-multiply equation (4.79) by S^{-1} to get the symmetric form

$$(S^{-1}AS)\,S^{-1}\widetilde{V} = \frac{\sinh\omega h}{\omega}\,(DS)^{-1}\,\widetilde{I}. \tag{4.85}$$

The second step in the procedure to construct the equivalent cable requires $S^{-1}AS$ to be transformed into a symmetric tri-diagonal matrix. This is achieved by observing that

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \dfrac{p}{\sqrt{p^2+q^2}} & 0 & \dfrac{q}{\sqrt{p^2+q^2}} \\ 0 & 0 & 1 & 0 \\ 0 & \dfrac{q}{\sqrt{p^2+q^2}} & 0 & -\dfrac{p}{\sqrt{p^2+q^2}} \end{bmatrix} \tag{4.86}$$

is a symmetric orthogonal matrix satisfying the property

$$H^{-1}(S^{-1}AS)H = T = \begin{bmatrix} -\cosh\omega h & \sqrt{p^2+q^2} & 0 & 0 \\ \sqrt{p^2+q^2} & -\cosh\omega h & \dfrac{pr}{\sqrt{p^2+q^2}} & 0 \\ 0 & \dfrac{pr}{\sqrt{p^2+q^2}} & -\cosh\omega h & \dfrac{qr}{\sqrt{p^2+q^2}} \\ 0 & 0 & \dfrac{qr}{\sqrt{p^2+q^2}} & -\cosh\omega h \end{bmatrix}. \tag{4.87}$$

The matrix H is called a Householder matrix, and its derivation will be described in Section 4.8.1. Since T is a symmetric tri-diagonal matrix, it may be interpreted as the symmetric form of a cable matrix. The second step is completed by the pre-multiplication of equation (4.85) by H^{-1} to obtain

$$T(SH)^{-1}\widetilde{V} = \frac{\sinh\omega h}{\omega}\,(DSH)^{-1}\,\widetilde{I}. \tag{4.88}$$

The final step of the procedure to construct the equivalent cable requires the derivation of the cable matrix $\mathcal{A}$, the symmetrising diagonal matrix $\mathcal{S}$ and the characteristic conductance for each section of the cable.

Extracting the equivalent cable

The off-diagonal entries of the equivalent cable matrix $\mathcal{A}$ and the symmetrising matrix $\mathcal{S}$ which transforms $\mathcal{A}$ into $T = \mathcal{S}^{-1}\mathcal{A}\mathcal{S}$ are extracted from T using the algorithm described in Section 4.4.8. The calculation advances through each row of T as follows

$$\begin{aligned}
\text{Row 1} \quad \mathcal{A}_{1,0} &= \frac{T^2_{0,1}}{\mathcal{A}_{0,1}} = \frac{c_1^2 + c_1c_3 + c_2c_3}{(c_1 + c_2)(c_1 + c_3)}, \\
\mathcal{A}_{1,2} &= 1 - \mathcal{A}_{1,0} = \frac{c_1c_2}{(c_1 + c_2)(c_1 + c_3)}, \\
\text{Row 2} \quad \mathcal{A}_{2,1} &= \frac{T^2_{1,2}}{\mathcal{A}_{1,2}} = \frac{c_1(c_1 + c_3)}{c_1^2 + c_1c_3 + c_2c_3}, \\
\mathcal{A}_{2,3} &= 1 - \mathcal{A}_{2,1} = \frac{c_2c_3}{c_1^2 + c_1c_3 + c_2c_3}, \\
\text{Row 3} \quad \mathcal{A}_{3,2} &= \frac{T^2_{2,3}}{\mathcal{A}_{2,3}} = 1.
\end{aligned} \tag{4.89}$$

As $\mathcal{A}_{3,2} = 1$, the last row of $\mathcal{A}$ confirms that the equivalent cable ends on a current injected terminal. The matrix $\mathcal{S}$ which symmetrises the equivalent cable has form

$$\mathcal{S} = \text{diag}\left[1, \sqrt{\frac{c_1^2 + c_1c_3 + c_2c_3}{(c_1 + c_2)(c_1 + c_3)}}, \sqrt{\frac{c_1 + c_3}{c_2}}, \frac{(c_1 + c_3)}{c_2}, \sqrt{\frac{c_1^2 + c_1c_3 + c_2c_3}{c_3(c_1 + c_3)}}\right]. \tag{4.90}$$

It is now straightforward matrix algebra to demonstrate that the voltage EGP matrix is

$$\Psi_V = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \dfrac{c_1}{c_1 + c_3} & 0 & \dfrac{c_3}{c_1 + c_3} \\ 0 & 0 & 1 & 0 \\ 0 & \dfrac{c_1 + c_2}{c_2} & 0 & -\dfrac{c_1}{c_2} \end{bmatrix} \tag{4.91}$$

and the current EGP matrix is

$$\Psi_C = \begin{bmatrix} \dfrac{d_1}{c_1 + c_3} & 0 & 0 & 0 \\ 0 & \dfrac{c_1(d_1 + d_2)}{(c_1 + c_3)(c_1 + c_2)} & 0 & \dfrac{d_1 + d_2}{c_1 + c_3} \\ 0 & 0 & \dfrac{d_2 + d_3}{c_2} & 0 \\ 0 & \dfrac{d_3}{c_2} & 0 & -\dfrac{c_1d_3}{c_2c_3} \end{bmatrix}. \tag{4.92}$$

For the equivalence of the branched and unbranched dendrites, it is essential that $\mathcal{I}_0(s) = I_0(s)$, namely that both dendrites have the same current flowing into the parent structure.

It therefore follows immediately from the first row of equation $\mathcal{I}_0(s) = \Psi_C I_0(s)$ that $d_1 = c_1 + c_3$. The definition (4.34) of the cable matrix $\mathcal{A}$ in terms of the characteristic conductances of the cable sections gives

$$\frac{d_1}{d_1 + d_2} = \frac{c_1^2 + c_1c_3 + c_2c_3}{(c_1 + c_3)(c_1 + c_2)}, \qquad \frac{d_2}{d_2 + d_3} = \frac{c_1(c_1 + c_3)}{c_1^2 + c_1c_3 + c_2c_3} \tag{4.93}$$

from which it follows by straightforward algebra that

$$d_1 = c_1 + c_3, \qquad d_2 = \frac{c_1c_2(c_1 + c_3)}{c_1^2 + c_1c_3 + c_2c_3} \qquad d_3 = \frac{c_2^2c_3}{c_1^2 + c_1c_3 + c_2c_3}. \tag{4.94}$$

Now that d_1, d_2 and d_3 are determined, expression (4.92) for the current EGP matrix Ψ_C can be expressed entirely in terms of the characteristic conductances of the branch dendrite to get

$$\Psi_C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \dfrac{c_1(c_1 + c_3)}{c_1^2 + c_1c_3 + c_2c_3} & 0 & \dfrac{(c_1 + c_2)(c_1 + c_3)}{c_1^2 + c_1c_3 + c_2c_3} \\ 0 & 0 & 1 & 0 \\ 0 & \dfrac{c_2c_3}{c_1^2 + c_1c_3 + c_2c_3} & 0 & -\dfrac{c_1c_2}{c_1^2 + c_1c_3 + c_2c_3} \end{bmatrix}. \tag{4.95}$$

Therefore the asymmetric Y-junction in Figure 4.5 with limbs of unequal electrotonic length $2h$ and h has an equivalent cable of electrotonic length $3h$ consisting of three uniform sections with characteristic conductances and current mappings

$$\text{Connected section 1} \quad \begin{cases} d_1 = c_1 + c_3 \\ \mathcal{I}_1 = \dfrac{c_1 + c_3}{c_1^2 + c_1c_3 + c_2c_3}\left[c_1I_1 + (c_1 + c_2)I_3\right], \end{cases}$$

$$\text{Connected section 2} \quad \begin{cases} d_2 = \dfrac{c_1c_2(c_1 + c_3)}{c_1^2 + c_1c_3 + c_2c_3} \\ \mathcal{I}_2 = I_2, \end{cases}$$

$$\text{Connected section 3} \quad \begin{cases} d_3 = \dfrac{c_2^2c_3}{c_1^2 + c_1c_3 + c_2c_3} \\ \mathcal{I}_3 = \dfrac{c_1c_2}{c_1^2 + c_1c_3 + c_2c_3}\left[I_1 - I_3\right]. \end{cases}$$

4.5.3 A symmetric Y-junction

Figure 4.6 illustrates a symmetric Y-junction consisting of two limbs with electrotonic length $2h$ and meeting at the branch point P_0. Currents $I_0(s), \ldots, I_4(s)$ are injected

into the Y-junction dendrite at the points $P_0, \ldots, P_4$ respectively. From the previous two examples, it is clear that this Y-junction conforms to Rall's first condition, namely that all dendritic terminals are the same electrotonic distance from the point of connection to the parent structure to the terminals. One outcome of this analysis will be to determine conditions under which this Y-junction has a Rall equivalent cylinder of electrotonic length $2h$, and when no such cylinder exists, to determine the equivalent cable of the Y-junction.

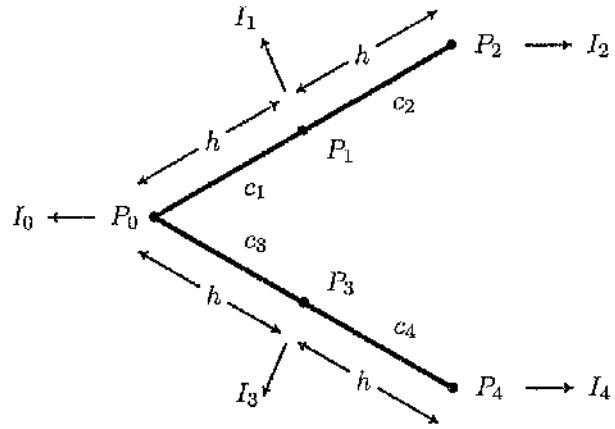

Figure 4.6: A Y-junction with limbs of electrotonic length $2h$. The sections joining P_0 to P_1, P_1 to P_2, P_0 to P_3 and P_3 to P_4 each have length h and characteristic conductances c_1, c_2, c_3 and c_4 respectively.

It follows from the application of particularised forms of equations (4.28) and (4.29) that the Laplace transforms of the membrane potentials $V_0(s), \ldots, V_4(s)$ at the points $P_0, \ldots, P_4$ respectively on the symmetric Y-junction illustrated in Figure 4.6 satisfy the algebraic equations

$$\begin{aligned}
-\widetilde{V}_0 \cosh \omega h + \frac{c_1}{c_1 + c_3} \widetilde{V}_1 + \frac{c_3}{c_1 + c_3} \widetilde{V}_3 &= \frac{\sinh \omega h}{\omega (c_1 + c_3)} \widetilde{I}_0 , \\
\frac{c_1}{c_1 + c_2} \widetilde{V}_0 - \widetilde{V}_1 \cosh \omega h + \frac{c_2}{c_1 + c_2} \widetilde{V}_2 &= \frac{\sinh \omega h}{\omega (c_1 + c_2)} \widetilde{I}_1 , \\
\widetilde{V}_1 - \widetilde{V}_2 \cosh \omega h &= \frac{\sinh \omega h}{\omega c_2} \widetilde{I}_2 , \\
\frac{c_3}{c_3 + c_4} \widetilde{V}_0 - \widetilde{V}_3 \cosh \omega h + \frac{c_4}{c_3 + c_4} \widetilde{V}_4 &= \frac{\sinh \omega h}{\omega (c_3 + c_4)} \widetilde{I}_3 , \\
\widetilde{V}_3 - \widetilde{V}_4 \cosh \omega h &= \frac{\sinh \omega h}{\omega c_4} \widetilde{I}_4 .
\end{aligned} \tag{4.96}$$

The system equations for the symmetric Y-junction therefore have matrix representation

$$A \widetilde{V} = \frac{\sinh \omega h}{\omega} D^{-1} \widetilde{I} \tag{4.97}$$

in which $\widetilde{V}$ is the vector of the Laplace transforms of the membrane potentials, $\widetilde{I}$ is the vector of the Laplace transforms of the injected currents and A is the tree matrix

$$\begin{bmatrix} -\cosh\omega h & \frac{c_1}{c_1+c_3} & 0 & \frac{c_3}{c_1+c_3} & 0 \\ \frac{c_1}{c_1+c_2} & -\cosh\omega h & \frac{c_2}{c_1+c_2} & 0 & 0 \\ 0 & 1 & -\cosh\omega h & 0 & 0 \\ \frac{c_3}{c_3+c_4} & 0 & 0 & -\cosh\omega h & \frac{c_4}{c_3+c_4} \\ 0 & 0 & 0 & 1 & -\cosh\omega h \end{bmatrix} . \tag{4.98}$$

The entries of the diagonal matrix D are the sums of the characteristic conductances of sections meeting at each point of the Y-junction. By following the procedure set out in Section 4.4.5 it can be shown that the diagonal matrix

$$S = \text{diag}\left[\, 1\,,\ \sqrt{\frac{c_1+c_3}{c_1+c_2}}\,,\ \sqrt{\frac{c_1+c_3}{c_2}}\,,\ \sqrt{\frac{c_1+c_3}{c_3+c_4}}\,,\ \sqrt{\frac{c_1+c_3}{c_4}}\,\right] \tag{4.99}$$

will symmetrise the tree matrix A to obtain

$$S^{-1}AS = \begin{bmatrix} -\cosh\omega h & p & 0 & q & 0 \\ p & -\cosh\omega h & r & 0 & 0 \\ 0 & r & -\cosh\omega h & 0 & 0 \\ q & 0 & 0 & -\cosh\omega h & w \\ 0 & 0 & 0 & w & -\cosh\omega h \end{bmatrix} \tag{4.100}$$

where p, q, r and w represent the expressions

$$\begin{aligned} p &= \frac{c_1}{\sqrt{(c_1+c_2)(c_1+c_3)}}\,, & q &= \frac{c_3}{\sqrt{(c_1+c_3)(c_3+c_4)}}\,, \\ r &= \sqrt{\frac{c_2}{c_1+c_2}}\,, & w &= \sqrt{\frac{c_4}{c_3+c_4}}\,. \end{aligned} \tag{4.101}$$

Pre-multiplication of equations (4.97) by S^{-1} reduces them to the symmetric form

$$(S^{-1}AS)S^{-1}\widetilde{V} = \frac{\sinh\omega h}{\omega}\,(DS)^{-1}\,\widetilde{I}\,. \tag{4.102}$$

Since $S^{-1}AS$ is symmetric but not tri-diagonal, the construction of the equivalent cable proceeds by tri-diagonalising $S^{-1}AS$. This is achieved by using the symmetric orthogonal

matrix

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \frac{p}{\sqrt{p^2+q^2}} & 0 & \frac{q}{\sqrt{p^2+q^2}} & 0 \\ 0 & 0 & \frac{pr}{\sqrt{p^2r^2+q^2w^2}} & 0 & \frac{qw}{\sqrt{p^2r^2+q^2w^2}} \\ 0 & \frac{q}{\sqrt{p^2+q^2}} & 0 & -\frac{p}{\sqrt{p^2+q^2}} & 0 \\ 0 & 0 & \frac{qw}{\sqrt{p^2r^2+q^2w^2}} & 0 & -\frac{pr}{\sqrt{p^2r^2+q^2w^2}} \end{bmatrix}. \quad (4.103)$$

For complex branched dendrites, H can be derived in a systematic way as a product of a finite sequence of Householder transformations. When equations (4.102) are pre-multiplied by H^{-1}, the result is

$$T(SH)^{-1}\widetilde{V} = \frac{\sinh\omega h}{\omega}(DSH)^{-1}\,\widetilde{I}, \quad (4.104)$$

where $T = (SH)^{-1}A(SH)$ is the symmetric tri-diagonal matrix

$$\begin{bmatrix} -\cosh\omega h & \sqrt{p^2+q^2} & 0 & 0 & 0 \\ \sqrt{p^2+q^2} & -\cosh\omega h & \sqrt{\frac{p^2r^2+q^2w^2}{p^2+q^2}} & 0 & 0 \\ 0 & \sqrt{\frac{p^2r^2+q^2w^2}{p^2+q^2}} & -\cosh\omega h & \frac{pq(r^2-w^2)}{\sqrt{p^2+q^2}\sqrt{p^2r^2+q^2w^2}} & 0 \\ 0 & 0 & \frac{pq(r^2-w^2)}{\sqrt{p^2+q^2}\sqrt{p^2r^2+q^2w^2}} & -\cosh\omega h & \frac{rw\sqrt{p^2+q^2}}{\sqrt{p^2r^2+q^2w^2}} \\ 0 & 0 & 0 & \frac{rw\sqrt{p^2+q^2}}{\sqrt{p^2r^2+q^2w^2}} & -\cosh\omega h \end{bmatrix}. \quad (4.105)$$

Equations (4.105) may be interpreted as the symmetrised, tri-diagonalised form of the system equations for the Y-junction in Figure 4.6, where T is the symmetric form of a cable matrix. The final stage in the construction of the equivalent cable involves the derivation of the cable matrix $\mathcal{A}$, the diagonal matrix $\mathcal{S}$ and the subsequent calculation of the characteristic conductances of the equivalent cable.

Extracting the equivalent cable

The off-diagonal entries of the equivalent cable matrix $\mathcal{A}$ and the symmetrising matrix $\mathcal{S}$ for which $T = \mathcal{S}^{-1}\mathcal{A}\mathcal{S}$ are extracted from T using the procedure set out in Section (4.4.8). The calculation now advances through each row of T on the assumption that $c_1c_4 \neq c_2c_3$.

The entries of the cable matrix $\mathcal{A}$ are calculated algebraically to obtain

$$\begin{aligned}
\text{Row 1}\quad \mathcal{A}_{1,0} &= \frac{T^2_{0,1}}{\mathcal{A}_{0,1}} = \frac{c_1^2(c_3+c_4)+c_3^2(c_1+c_2)}{(c_1+c_2)(c_1+c_3)(c_3+c_4)}\,,\\
\mathcal{A}_{1,2} &= 1-\mathcal{A}_{1,0} = \frac{c_1c_2(c_3+c_4)+c_3c_4(c_1+c_2)}{(c_1+c_2)(c_1+c_3)(c_3+c_4)}\,,\\
\text{Row 2}\quad \mathcal{A}_{2,1} &= \frac{T^2_{1,2}}{\mathcal{A}_{1,2}} = \frac{(c_1+c_3)[c_1^2c_2(c_3+c_4)^2+c_3^2c_4(c_1+c_2)^2]}{[c_1^2(c_3+c_4)+c_3^2(c_1+c_2)][c_1c_2(c_3+c_4)+c_3c_4(c_1+c_2)]}\,,\\
\mathcal{A}_{2,3} &= 1-\mathcal{A}_{2,1} = \frac{c_1c_3(c_2c_3-c_1c_4)^2}{[c_1^2(c_3+c_4)+c_3^2(c_1+c_2)][c_1c_2(c_3+c_4)+c_3c_4(c_1+c_2)]}\,,\\
\text{Row 3}\quad \mathcal{A}_{3,2} &= \frac{T^2_{2,3}}{\mathcal{A}_{2,3}} = \frac{c_1c_3[c_1c_2(c_3+c_4)+c_3c_4(c_1+c_2)]}{c_1^2c_2(c_3+c_4)^2+c_3^2c_4(c_1+c_2)^2}\,,\\
\mathcal{A}_{3,4} &= 1-\mathcal{A}_{3,2} = \frac{c_2c_4[c_1^2(c_3+c_4)+c_3^2(c_1+c_2)]}{c_1^2c_2(c_3+c_4)^2+c_3^2c_4(c_1+c_2)^2}\,,\\
\text{Row 4}\quad \mathcal{A}_{4,3} &= \frac{T^2_{3,4}}{\mathcal{A}_{3,4}} = 1\,.
\end{aligned} \tag{4.106}$$

When $c_1c_4 \neq c_2c_3$, the equivalent cable consists of four sections and terminates in a current injected terminal. The matrix $\mathcal{S}$ mapping the equivalent cable into the symmetrised Y-junction is determined from equation (4.38) and takes the value

$$\begin{aligned}
\mathcal{S} = \operatorname{diag}\Bigg[\,1\,,\ &\sqrt{\frac{c_1^2(c_3+c_4)+c_3^2(c_1+c_2)}{(c_1+c_2)(c_1+c_3)(c_3+c_4)}}\,,\ \sqrt{\frac{[c_1+c_3][c_1^2c_2(c_3+c_4)^2+c_3^2c_4(c_1+c_2)^2]}{[c_1c_2(c_3+c_4)+c_3c_4(c_1+c_2)]^2}}\,,\\
&\sqrt{\frac{[c_1+c_3][c_1^2(c_3+c_4)+c_3^2(c_1+c_2)]}{[c_2c_3-c_1c_4]^2}}\,,\ \sqrt{\frac{[c_1+c_3][c_1^2c_2(c_3+c_4)^2+c_3^2c_4(c_1+c_2)^2]}{c_2c_4[c_2c_3-c_1c_4]^2}}\,\Bigg]\,.
\end{aligned} \tag{4.107}$$

Given $\mathcal{S}$, it is straightforward matrix algebra to show that the voltage EGP matrix is

$$\Psi_{\mathrm{V}} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0\\
0 & \dfrac{c_1}{c_1+c_3} & 0 & \dfrac{c_3}{c_1+c_3} & 0\\
0 & 0 & \dfrac{c_1c_2(c_3+c_4)}{\eta} & 0 & \dfrac{c_3c_4(c_1+c_2)}{\eta}\\
0 & \dfrac{c_3(c_1+c_2)}{c_2c_3-c_1c_4} & 0 & \dfrac{c_1(c_3+c_4)}{c_2c_3-c_1c_4} & 0\\
0 & 0 & \dfrac{c_3(c_1+c_2)}{c_2c_3-c_1c_4} & 0 & \dfrac{c_1(c_3+c_4)}{c_2c_3-c_1c_4}
\end{bmatrix} \tag{4.108}$$

where $\eta = c_1c_2(c_3+c_4)+c_3c_4(c_1+c_2)$. The characteristic conductances of the sections of the equivalent cable are determined directly from $\mathcal{A}$, and take the values

$$d_1 = c_1 + c_3\,, \qquad d_2 = (c_1+c_3)\left[\frac{c_1c_2(c_3+c_4)+c_3c_4(c_1+c_2)}{c_1^2(c_3+c_4)+c_3^2(c_1+c_2)}\right],$$

$$d_3 = \frac{c_1c_3\,[c_2c_3-c_1c_4]^2\,[c_1c_2(c_3+c_4)+c_3c_4(c_1+c_2)]}{[c_1^2(c_3+c_4)+c_3^2(c_1+c_2)]\,[c_1^2c_2(c_3+c_4)^2+c_3^2c_4(c_1+c_2)^2]}\,, \tag{4.109}$$

$$d_4 = \frac{c_2c_4[c_2c_3-c_1c_4]^2}{c_1^2c_2(c_3+c_4)^2+c_3^2c_4(c_1+c_2)^2}\,.$$

Given the values $d_1,\ldots,d_4$ in (4.109), the current EGP matrix $\Psi_C = \mathcal{D}\,\Psi_V\,D^{-1}$ may be expressed in terms of $c_1,\ldots,c_4$. Since $\mathcal{D}$ and D^{-1} are diagonal matrices then Ψ_C and Ψ_V are structurally identical. By this it is meant, that the non-zero entries of Ψ_C and Ψ_V are in identical locations. Therefore in this example, Ψ_C has an identical structural form to Ψ_V therefore follows that the current input at a given location on the branch structure maps to locations on the equivalent cable that are no closer to its soma than that of the input on the branched structure.

Special case: $c_1c_4 = c_2c_3$

This special case corresponds to a Y-junction with limbs satisfying the Rall condition $c_1/c_2 = c_3/c_4$. Here, one limb of the Y-junction is a scaled version of the other limb. Therefore, in the special case in which $r = w$, or equivalently $c_1c_4 = c_2c_3$, the tri-diagonal matrix T in equations (4.104) takes the particularly simple form

$$T = \left[\begin{array}{ccc|cc} -\cosh\omega h & \sqrt{p^2+q^2} & 0 & 0 & 0 \\ \sqrt{p^2+q^2} & -\cosh\omega h & r & 0 & 0 \\ 0 & r & -\cosh\omega h & 0 & 0 \\ \hline 0 & 0 & 0 & -\cosh\omega h & r \\ 0 & 0 & 0 & r & -\cosh\omega h \end{array}\right]. \tag{4.110}$$

The block diagonal form of T forces the construction of the equivalent cable to proceed in two stages, the first dealing with the tri-diagonal matrix T_1 defined by the upper 3×3 block matrix in T, and the second dealing with the tri-diagonal matrix T_2 defined by the lower 2×2 matrix in T. Thus equations (4.104) decompose into the independent sets of equations

$$T_1M_1\widetilde{V} = \frac{\sinh\omega h}{\omega}\,R_1\,\widetilde{I}\,, \qquad T_2M_2\widetilde{V} = \frac{\sinh\omega h}{\omega}\,R_2\,\widetilde{I} \tag{4.111}$$

in which M_1 and M_2 are respectively the 3×5 and 2×5 matrices

$$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{c_1(c_1+c_2)}}{c_1+c_3} & 0 & \frac{\sqrt{c_3(c_3+c_4)}}{c_1+c_3} & 0 \\ 0 & 0 & \frac{\sqrt{c_1 c_2}}{c_1+c_3} & 0 & \frac{\sqrt{c_3 c_4}}{c_1+c_3} \end{bmatrix},$$

$$M_2 = \begin{bmatrix} 0 & \frac{\sqrt{c_3(c_1+c_2)}}{c_1+c_3} & 0 & \frac{\sqrt{c_1(c_3+c_4)}}{c_1+c_3} & 0 \\ 0 & 0 & \frac{\sqrt{c_2 c_3}}{c_1+c_3} & 0 & -\frac{\sqrt{c_1 c_4}}{c_1+c_3} \end{bmatrix}, \tag{4.112}$$

and R_1 and R_2 are respectively the 3×5 and 2×5 matrices

$$R_1 = \frac{1}{c_1+c_3} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \sqrt{\frac{c_1}{c_1+c_2}} & 0 & \sqrt{\frac{c_3}{c_3+c_4}} & 0 \\ 0 & 0 & \sqrt{\frac{c_1}{c_2}} & 0 & \sqrt{\frac{c_3}{c_4}} \end{bmatrix},$$

$$R_2 = \frac{1}{c_1+c_3} \begin{bmatrix} 0 & \sqrt{\frac{c_3}{c_1+c_2}} & 0 & -\sqrt{\frac{c_1}{c_3+c_4}} & 0 \\ 0 & 0 & \sqrt{\frac{c_3}{c_2}} & 0 & -\sqrt{\frac{c_1}{c_4}} \end{bmatrix}. \tag{4.113}$$

The matrices M_1 and M_2 are formed respectively from the first three rows and last two rows of $(SH)^{-1}$, while R_1 and R_2 are likewise formed respectively from the first three rows and last two rows of $(DSH)^{-1}$. Each set of equations in (4.111) represents a different component of the equivalent cable, the first is connected to the parent structure whereas the second is not.

Connected cable

The entries of the cable matrix $\mathcal{A}$ corresponding to the first of equations (4.111) are now calculated sequentially. With the tri-diagonal matrix T_1 represented by T (for convenience), this calculation gives

$$\begin{aligned} \text{Row 1} \quad \mathcal{A}_{1,0} &= \frac{T^2_{0,1}}{\mathcal{A}_{0,1}} = \frac{c_1}{c_1+c_2} = \frac{c_3}{c_3+c_4}, \\ \mathcal{A}_{1,2} &= 1 - \mathcal{A}_{1,0} = \frac{c_2}{c_1+c_2} = \frac{c_4}{c_3+c_4}, \\ \text{Row 2} \quad \mathcal{A}_{2,1} &= \frac{T^2_{1,2}}{\mathcal{A}_{1,2}} = 1 \end{aligned} \tag{4.114}$$

where the transformation from $\mathcal{A}$ to T is effected with the diagonal matrix

$$\mathcal{S}_1 = \left[1 \, , \, \sqrt{\frac{c_3}{c_3 + c_4}} \, , \, \sqrt{\frac{c_3}{c_4}} \right] . \tag{4.115}$$

In this instance the expressions for the voltage and current EGP matrices corresponding to formulae (4.50) are respectively

$$\begin{aligned} \Psi_V = \mathcal{S}_1 M_1 &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \dfrac{c_1}{c_1 + c_3} & 0 & \dfrac{c_3}{c_1 + c_3} & 0 \\ 0 & 0 & \dfrac{c_1}{c_1 + c_3} & 0 & \dfrac{c_3}{c_1 + c_3} \end{bmatrix} , \\ \Psi_C = \mathcal{D} \mathcal{S}_1 R_1 &= \frac{d_1}{c_1 + c_3} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} . \end{aligned} \tag{4.116}$$

Since $\mathcal{I}_0 = I_0$, then $d_1 = c_1 + c_3$ and the expression for $\mathcal{A}_{0,1}$ leads to $d_2 = c_2(c_1 + c_3)/c_1$. Cable potentials and injected currents $\mathcal{V}$ and $\mathcal{I}$ are related to tree potentials and currents V and I by the respective formulae $\mathcal{V} = \Psi_V V$ and $\mathcal{I} = \Psi_C I$. These relationships have component form

$$\begin{aligned} \mathcal{V}_0 &= V_0 \, , & \mathcal{I}_0 &= I_0 \, , \\ \mathcal{V}_1 &= \frac{c_1 V_1 + c_3 V_3}{c_1 + c_3} \, , & \mathcal{I}_1 &= I_1 + I_3 \, , \\ \mathcal{V}_2 &= \frac{c_1 V_2 + c_3 V_4}{c_1 + c_3} \, , & \mathcal{I}_2 &= I_2 + I_4 \, . \end{aligned} \tag{4.117}$$

Disconnected cable

In this section the currents $\mathcal{I}_0$, $\mathcal{I}_1$ and $\mathcal{I}_2$ and the potentials $\mathcal{V}_0$, $\mathcal{V}_1$ and $\mathcal{V}_2$ refer to the left-hand node, centre node and right-hand node respectively of the disconnected cable. The entries of the associated cable matrix $\mathcal{A}$ corresponding to the second of equations (4.111) are now calculated sequentially. With the tri-diagonal matrix T_2 represented by T (for convenience), this calculation gives

$$\begin{aligned} \text{Row 1} \quad \mathcal{A}_{1,0} &= \frac{T^2_{0,1}}{\mathcal{A}_{0,1}} = \frac{c_2}{c_1 + c_2} = \frac{c_4}{c_3 + c_4} \, , \\ \mathcal{A}_{1,2} &= 1 - \mathcal{A}_{1,0} = \frac{c_1}{c_1 + c_2} = \frac{c_3}{c_3 + c_4} \end{aligned} \tag{4.118}$$

where transformation from $\mathcal{A}$ to T is effected with the diagonal matrix

$$\mathcal{S}_2 = \left[1 \, , \, \sqrt{\frac{c_2}{c_1 + c_2}} \right] . \tag{4.119}$$

In this instance the expressions for the voltage and current EGP matrices corresponding to formulae (4.50) are respectively

$$\begin{aligned}
\Psi_V = \mathcal{S}_2 M_2 &= \frac{\sqrt{c_3(c_1+c_2)}}{c_1+c_3}\begin{bmatrix} 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & \dfrac{c_2}{c_1+c_2} & 0 & -\dfrac{c_2}{c_1+c_2} \end{bmatrix}, \\
\Psi_C = \mathcal{D}\mathcal{S}_2 R_2 &= \frac{d_1\sqrt{c_3(c_1+c_2)}}{c_2(c_1+c_3)}\begin{bmatrix} 0 & \dfrac{c_2}{c_1+c_2} & 0 & -\dfrac{c_2}{c_3+c_4} & 0 \\ 0 & 0 & 1 & 0 & -\dfrac{c_1}{c_3} \end{bmatrix}.
\end{aligned} \tag{4.120}$$

Note that Ψ_V and Ψ_C only give the potential and injected currents at the first two nodes of the detached cable. Specifically,

$$\begin{aligned}
\mathcal{V}_0 &= \frac{\sqrt{c_3(c_1+c_2)}}{c_1+c_3}\left[V_1 - V_3\right], \\
\mathcal{V}_1 &= \frac{c_2}{c_1+c_3}\sqrt{\frac{c_3}{c_1+c_2}}\left[V_2 - V_4\right], \\
\mathcal{I}_0 &= \frac{d_1}{c_1+c_3}\sqrt{\frac{c_3}{c_1+c_2}}\left[I_1 - \frac{c_1}{c_3}I_2\right], \\
\mathcal{I}_1 &= \frac{d_1}{c_1+c_3}\frac{\sqrt{c_3(c_1+c_2)}}{c_2}\left[I_2 - \frac{c_1}{c_3}I_4\right].
\end{aligned} \tag{4.121}$$

In the expressions for $\mathcal{I}_0$ and $\mathcal{I}_1$, the characteristic conductance of the first section of the detached cable is indeterminate unlike the first section of the connected cable.

It can be demonstrated that $\mathcal{V}_2 = 0$ and that the equation contributed by the third node on the detached cable determines the injected current required to maintain $\mathcal{V}_2 = 0$. As discussed in the derivation of the model, the third equation does not appear in the mathematical formulation of the cable based on the determination of unknown potentials. Specifically, although the mathematical description of the detached cable contains three nodes and three unknown functions, only two of these are unknown potentials; the third is an injected current and so does not feature in a matrix representation of the detached cable based on unknown potentials.

In this example, the symmetric Y-junction in Figure 4.6 with limbs of length $2h$ forms an equivalent cable with electrotonic length $4h$. The first pair of uniform sections of the cable are connected to the parent structure while the second pair are disconnected from it.

4.6 Application of the equivalent cable to spinal interneurons

The development of the equivalent cable and its associated mapping allows studies into the influence of complex dendritic morphology to be extended beyond those based on approximate representations of dendritic morphology (see Vetter, Roth and Häusser, 2001; Mainen and Sejnowski, 1996). This section will describe the generation of equivalent cables for spinal interneurons and the estimation of the distribution of associated synaptic contacts when mapped to the equivalent cable.

The conventional description of synaptic location employs a Sholl analysis (Olave *et al.*, 2002). This procedure counts the number of contacts falling on regions of dendrite contained within concentric shells with radii that increase in 25 μm steps from the centre of the soma. This procedure is applied to a collection of cells. For these cells a histogram of the number of contacts within each concentric shell is calculated. The Sholl procedure inherently assumes that dendrites radiate outwards from the soma, and therefore when this does not happen, the procedure misrepresents the density of contacts. That is, a distal contact on a branch that turns back towards the soma may be represented as a proximal contact. Furthermore, the histogram process is a blunt tool making it difficult to draw conclusions on the distributions of contacts on the same type of cells or between different types of cells. The mapping derived in the construction of the equivalent cable provides a means to investigate individual neurons and synaptic contacts in a way that incorporates fully the morphology and biophysical properties of the cell without resorting to a histogram procedure.

4.6.1 Distribution of contacts

The equivalent cables of two cholinergic interneurons located in laminae III/IV of the dorsal horn of the spinal cord have been generated numerically. An overview of the neurons and their input structure is given in Chapter 3, whilst a detailed description of these neurons can be found in Olave *et al.* (2002). To my knowledge, the following examples are the first examples of *equivalent cables* constructed from real neurons by contrast with those constructed by piecewise empirical methods (*e.g.* Segev and Burke, 1989).

The procedure uses the contact location from the Neurolucida files and assigns a strength

of one unit to each contact. A discretisation interval of one-thousandth of an electrotonic unit is used in the construction of each sample dendrite, resulting in 1300-1400 nodes per dendrite, and a placement of contacts with a maximum error of one two-thousandth of an electrotonic unit. The EGP matrix, Ψ_C, described in Section 4.4.7 is now used to map contacts I on the branched dendrite to contacts $\mathcal{I}$ on the equivalent cable according to the rule that $\mathcal{I} = \Psi_C I$. The fraction of $\mathcal{I}$ received by the connected cable up to and including node k is formed by summing the entries of $\mathcal{I}$ from $j = 0$ to $j = k$ and then normalised by dividing this sum by the total number of contacts on the dendrite. The function is smoothed by interpolation (see Section 4.8.2), and its derivative with respect to electrotonic distance from the soma gives the density of contacts at each location on the equivalent cable.

As the mapping of contacts from the branched model to the equivalent cable is unique, comparing the contact density for different classes of contacts on the equivalent cable is equivalent to comparing the distribution of these contacts on the branched model. This procedure has clear benefits over the Sholl analysis, in particular it is unaffected by the phenomenon of branches turning back towards the soma, giving a false impression of the location of synapses.

Example 1: Equivalent cable representation of myelinated afferent input to cholinergic interneurons

Figure 4.7A shows a typical lamina III/IV spinal interneuron receiving myelinated afferent input. The dendogram and equivalent cable for this cell can be seen in Figures 4.7B and 4.7C respectively. Note that this equivalent cable has an electrotonic length of 1.39 eu. The cumulative distribution of contacts shown in Figure 4.7D indicates that almost 50% of the combined effect of the distribution of contacts on the branched structure lies within 0.1 eu of the soma. The plot of contact density in Figure 4.7E suggests that the influence of the contacts on the soma declines steadily with increasing distance from the soma until disappearing at approximately 0.6 eu. At 0.6 eu, the diameter of the equivalent cable has fallen to roughly 2% of its initial value.

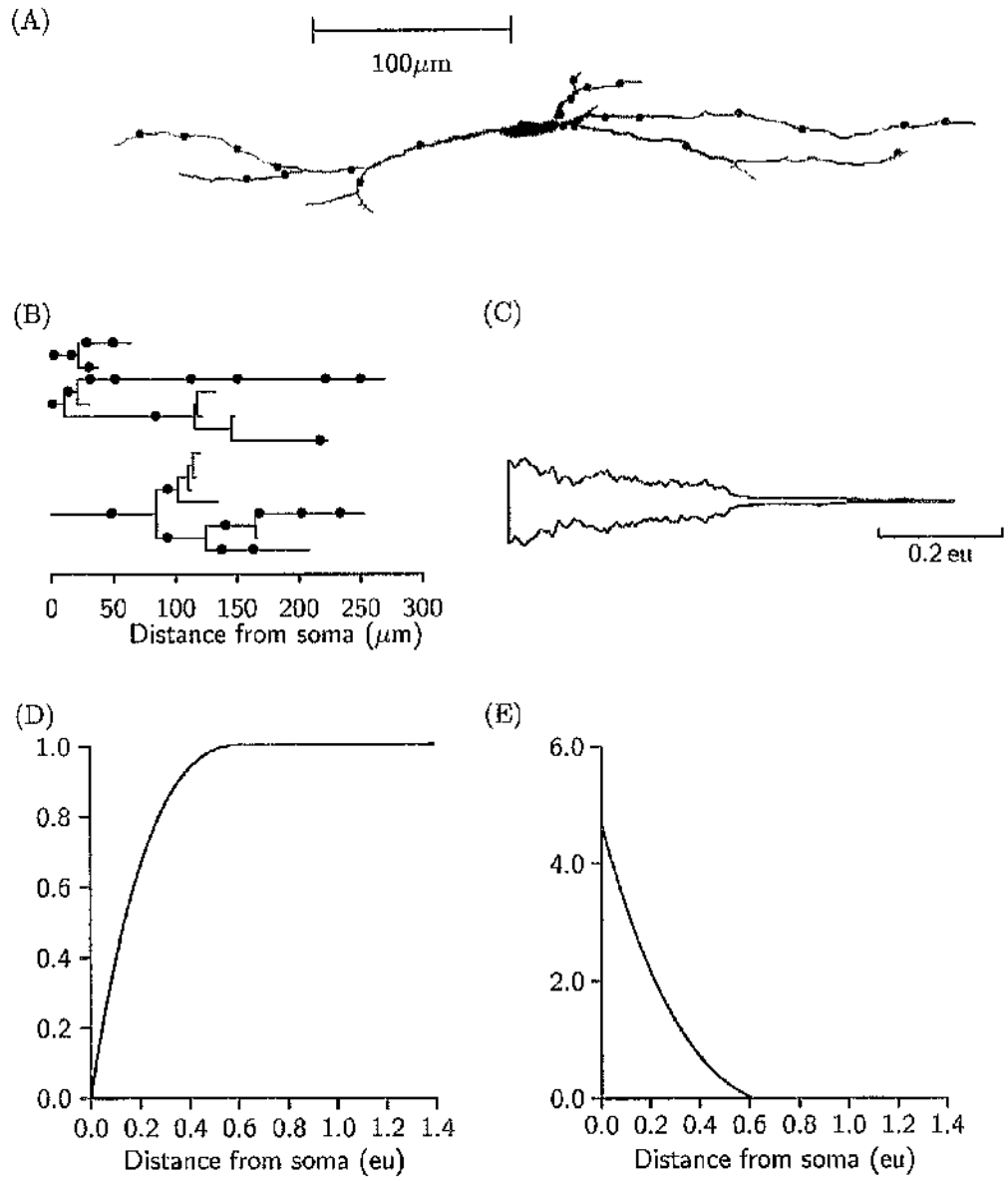

Figure 4.7: An example of a cholinergic interneuron (A) which receives myelinated primary afferent input (•) with its associated dendogram (B) and equivalent cable (C). The first 0.72 eu of the connected section of the equivalent cable is shown, the full cable has length 1.39 eu. The cumulative strength of the contacts (D) and their associated density (E) for the cell are also illustrated.

Example 2: Equivalent cable representation of unmyelinated afferent input to cholinergic interneurons

Figure 4.8A shows a typical lamina III/IV spinal interneuron receiving three distinct classes of unmyelinated afferent input. Type 1 contacts are non-peptidergic primary afferents (IB4

staining), type 2 contacts are peptidergic primary afferents (CGRP staining) and type 0 contacts are a rarer class which stain for both IB4 and CGRP. In Figure 4.8A, the three classes of contacts can be seen (type 0 - ★, type 1 - • and type 2 - ▲). The dendogram and equivalent cable for this cell can be seen in Figures 4.8B and 4.8C respectively.

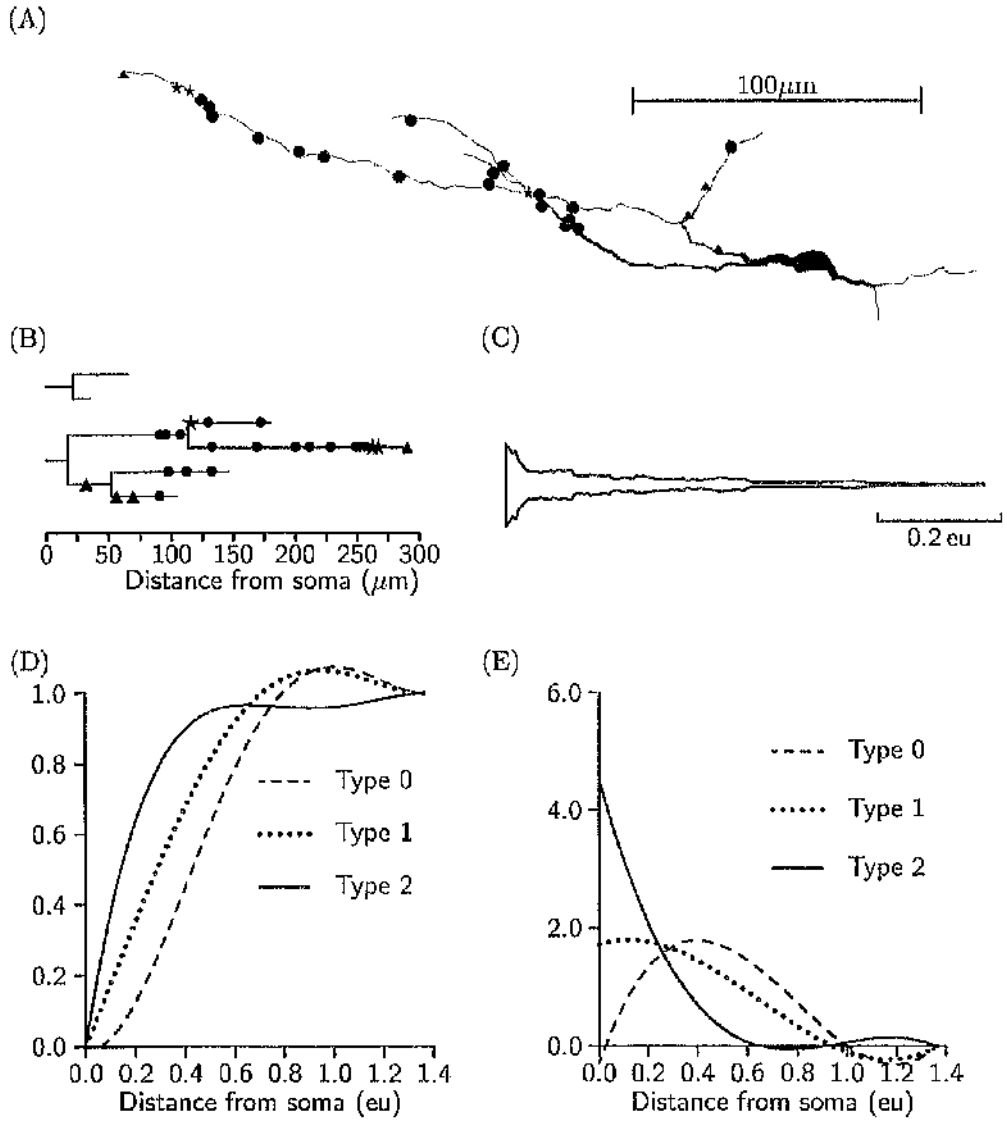

Figure 4.8: An example of a cholinergic interneuron (A) which receives unmyelinated primary afferents (type 0 - ★, type 1 - • and type 2 - ▲) with its associated dendogram (B) and equivalent cable (C). The first 0.77 eu of the connected section of the equivalent cable is shown, the full cable has length 1.36 eu. The cumulative strength of contacts (D) and their associated density (E) for the cell are also illustrated.

The equivalent cable has electrotonic length 1.36 eu, and has a distinctly different shape to that extracted from the interneuron receiving myelinated afferent input (see Figure 4.7B). The cumulative strength of contacts and the associated contact density are calculated for each class of contact. The cumulative strengths (Figure 4.8D) for each class of contact suggest that the contacts form different distributions along the equivalent cable. The plots of contact density (Figure 4.8E) accentuate the differences between the effects of the three classes of contact. Close to the soma, at less than 0.3 eu, the effect of the Type-2 contacts is dominant followed by Type-1 contacts which are in turn stronger than the Type-0 contacts. At intermediate distances, approximately 0.3 - 1.0 eu, the effects of the contacts changes such that Type-0 contacts are stronger than Type-1 which are in turn stronger than Type-2 contacts. The plots of contact density clearly define the distinct regions over which each class of contact has its effect. A comparison of the distributions of contacts from myelinated and unmyelinated afferents indicates that Type-2 unmyelinated contacts are almost distributed identically to myelinated contacts.

The application of the equivalent cable procedure to real neurons has demonstrated first, that the equivalent cables for spinal interneurons receiving myelinated afferent input form distinctly different shaped cables than those receiving unmyelinated afferent input, second, that the distribution of contacts varies with the type of afferent input, and finally, that there are three distinct distributions of unmyelinated afferent input. The second result confirms the observation by Olave *et al.* (2002) that myelinated and unmyelinated afferent input form different distributions, however, the final result extends the conclusions of Olave *et al.* (2002) by demonstrating that the distributions of the unmyelinated inputs are themselves different.

4.7 Conclusions

The equivalent cable of a branched dendrite is an unbranched representation of the dendrite and its input. The response at the soma of the branched dendrite to any configuration of input can be represented exactly at the soma of the equivalent cable. Conversely, the response at the soma to any configuration of input on the cable can be represented exactly by the response at the soma of the branched dendrite. This chapter has developed a novel analytical procedure for constructing an equivalent cable. This procedure transforms an arbitrarily branched dendrite with arbitrary configurations of input into a piecewise uniform cable with an input structure determined uniquely by the configuration of inputs on the original tree. The cable is generated about the point of contact of the dendrite with its parent structure and may be accompanied by electrically isolated cables which are disconnected from both the parent structure and each other. The procedure for constructing the equivalent cable ensures that the electrotonic length of the connected cable and any disconnected cables equals the total electrotonic length of the dendrite as defined by the sum of the electrotonic length of all its segments. It is shown that the matrix corresponding to an arbitrarily branched dendrite can be transformed to a symmetric tri-diagonal matrix, which can then be associated with the canonical form of a cable. By this route, the representation of the original branched dendrite by piecewise uniform sections is mapped into the representation of a cable (the equivalent cable) with piecewise uniform sections. The construction process specifies how potentials and input on the branched dendrite is mapped bijectively to potentials and input on the equivalent cable.

This procedure for constructing the equivalent cable was used to characterise contacts on spinal interneurons. This method could be applied to single interneurons and proved to be more powerful than the traditional method based on the Sholl analysis.

4.8 Mathematical appendix

4.8.1 Houscholder procedure

The Householder procedure is here used to reduce a symmetric tree matrix to a symmetric tri-diagonal matrix (Golub and van Loan, 1989) as part of the procedure to construct an equivalent cable. The resulting symmetric tri-diagonal matrix can then be interpreted as a symmetric cable matrix. Previously, the equivalent cable has been developed using the Lanczos procedure (Ogden *et al.*, 1999) to transform the symmetric tree matrix into the tri-diagonal cable matrix. Setting aside the fact that the Lanczos procedure is based on a numerical approximation of the cable equation using central differences, the Householder approach enjoys two advantages over the Lanczos procedure. First, it is numerically stable by contrast with the Lanczos procedure which is well recognised to suffer from the effects of rounding error (Golub and van Loan, 1989). Second, the Lanczos procedure often fails to develop the complete symmetric tri-diagonal matrix in a single operation unlike the Householder algorithm which always develops the complete symmetric tri-diagonal matrix.

Householder matrices

Given any unit column vector U of dimension n, the *Householder matrix* H (see Golub and Van Loan 1989) is defined by

$$H = I - 2U\,U^T \tag{4.122}$$

where I is the $n \times n$ identity matrix. By construction, the matrix H is symmetric and orthogonal. While the symmetry of H is obvious, the orthogonality property follows from the calculation

$$\begin{aligned} H^2 &= (I - 2U\,U^T)(I - 2U\,U^T) \\ &= I - 4U\,U^T + 4U\,(U^T U)\,U^T \\ &= I - 4U\,U^T + 4U\,U^T = I. \end{aligned} \tag{4.123}$$

Thus $H = H^T = H^{-1}$. Given any symmetric $(n+1) \times (n+1)$ tree matrix $S^{-1}AS$, there is a sequence of $(n-1)$ orthogonal matrices $Q_1, Q_2, \ldots, Q_{n-1}$ such that

$$(Q_{n-1}^{-1} \cdots Q_1^{-1})(S^{-1}AS)(Q_1 \cdots Q_{n-1}) = T \tag{4.124}$$

where T is a symmetric tri-diagonal matrix (see Golub and Van Loan, 1989). To interpret the final tri-diagonal form of the tree matrix as a cable attached to the parent structure, it is essential for the Householder procedure to start with the row of the symmetrised tree matrix corresponding to its point of connection to the parent structure, *i.e.* P_0.

Let the orthogonal matrix Q_1 and the symmetric tree matrix $W = S^{-1}AS$ have respective block matrix forms

$$W = \begin{bmatrix} w_{00} & Y^T \\ Y & Z \end{bmatrix}, \qquad Q_1 = \begin{bmatrix} I_1 & 0 \\ 0 & H_1 \end{bmatrix}, \tag{4.125}$$

where Y is a column vector of dimension n, Z is a symmetric $n \times n$ matrix and H_1 is an $n \times n$ Householder matrix constructed from a unit vector U. Assuming that the first row and column of W are not already in tri-diagonal form, the specification of U in the construction of H_1 is motivated by the result

$$Q_1^{-1}WQ_1 = \begin{bmatrix} w_{00} & (H_1Y)^T \\ \\ H_1Y & H_1^TZH_1 \end{bmatrix}.$$

The vector U is chosen to ensure that all elements of the column vector H_1Y are zero except the first element. If this is possible, the first row and column of $Q_1^{-1}WQ_1$ will form the first row and column of a tri-diagonal matrix. Furthermore, $H_1^TZH_1$ is itself an $n \times n$ symmetric matrix which assumes the role of W in the next step of the Householder procedure. This algorithm proceeds iteratively for $(n-1)$ steps, finally generating a 2×2 matrix $H_{n-1}^TZH_{n-1}$ on the last iteration. It can be shown that the choice

$$U = \frac{Y + \alpha|Y|E_1}{\sqrt{2|Y|(|Y| + \alpha Y_1)}}, \qquad \begin{aligned} E_1 &= [1, 0, \cdots (n-1) \text{ times } \cdots]^T, \\ Y_1 &= Y^TE_1, \end{aligned} \tag{4.126}$$

with $\alpha^2 = 1$ defines an H_1 with the property that $H_1Y = -\alpha|Y|E_1$, that is, the entries of H_1Y are all zero except the first entry. This property of H_1 can be established by elementary matrix algebra. The stability of the Householder procedure is guaranteed by setting $\alpha = 1$ if $Y_1 \geq 0$ and $\alpha = -1$ if $Y_1 < 0$, that is, α is conventionally chosen to make αY_1 non-negative.

Once H_1 is known, the symmetric $n \times n$ matrix $H_1^TZH_1$ is computed and the entire procedure repeated using the $(n+1) \times (n+1)$ orthogonal matrix Q_2 with block form

$$Q_2 = \begin{bmatrix} I_2 & 0 \\ 0 & H_2 \end{bmatrix}$$

in which H_2 is an $(n-1)\times(n-1)$ Householder matrix. Continued repetition of this procedure generates a sequence of orthogonal matrices $Q_1, Q_2, \cdots, Q_{n-1}$ such that

$$(Q_{n-1}\cdots Q_k\cdots Q_1)(S^{-1}AS)(Q_1\cdots Q_k\cdots Q_{n-1}) = T \tag{4.127}$$

where T is a symmetric tri-diagonal matrix to be interpreted as the symmetrised matrix of an equivalent cable. In order to construct the mapping of injected current on the branched dendrite to its equivalent cable and *vice-versa*, it is necessary to know the orthogonal matrix $Q = Q_1, Q_2, \cdots, Q_{n-1}$. In practice, this matrix can be computed efficiently by recognising that the original symmetrised tree matrix can be systematically overwritten as Q is constructed *provided* the calculation is performed backwards, that is, the calculation begins with Q_{n-1} and ends with Q_1. Using this strategy, it is never necessary to store the Householder matrices $H_1, \cdots, H_{n-1}$.

A numerical example

A more transparent picture of this procedure can be established by considering the reduction of the symmetric matrix

$$W = \begin{bmatrix} 9 & -1 & 2 & 2 \\ -1 & 3 & 4 & 2 \\ 2 & 4 & 14 & -3 \\ 2 & 2 & -3 & 4 \end{bmatrix} \tag{4.128}$$

to symmetric tri-diagonal form using Householder matrices. From the block matrices (4.125) we can see that $Y^T = (-1,2,2)$, $Y = (-1,2,2)^T$ and therefore $|Y| = 3$. The vector U is defined as

$$U = (-1,2,2)^T + \alpha|Y|(1,0,0)^T, \qquad \alpha^2 = 1$$

where $\alpha = -1$ in this case since the first element is negative. Therefore $U = (-4,2,2)^T$ and the Householder matrix defined by U is therefore

$$H = I - \frac{2}{|U|^2}UU^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \frac{2}{24}\begin{bmatrix} 16 & -8 & -8 \\ -8 & 4 & 4 \\ -8 & 4 & 4 \end{bmatrix} = \frac{1}{3}\begin{bmatrix} -1 & 2 & 2 \\ 2 & 2 & -1 \\ 2 & -1 & 2 \end{bmatrix}.$$

From the definitions (4.125), I now define

$$P_1 = \begin{bmatrix} I_1 & 0 \\ 0 & H \end{bmatrix} = \frac{1}{3}\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & -1 & 2 & 2 \\ 0 & 2 & 2 & -1 \\ 0 & 2 & -1 & 2 \end{bmatrix},$$

which is used to compute $P_1^{-1}WP_1$ where W is given by (4.128). It is an easy calculation to show that

$$P_1^{-1}WP_1 = \frac{1}{9}\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & -1 & 2 & 2 \\ 0 & 2 & 2 & -1 \\ 0 & 2 & -1 & 2 \end{bmatrix}\begin{bmatrix} 9 & -1 & 2 & 2 \\ -1 & 3 & 4 & 2 \\ 2 & 4 & 14 & -3 \\ 2 & 2 & -3 & 4 \end{bmatrix}\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & -1 & 2 & 2 \\ 0 & 2 & 2 & -1 \\ 0 & 2 & -1 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 9 & 3 & 0 & 0 \\ 3 & 3 & 6 & 0 \\ 0 & 6 & 12 & -3 \\ 0 & 0 & -3 & 6 \end{bmatrix}.$$

In this instance, one step of the Householder procedure has neatly transformed a symmetric matrix into a symmetric tri-diagonal matrix.

4.8.2 Estimating the density of contacts

Let the equivalent cable have electrotonic length L and suppose that $F(x)$ is the cumulative sum of the strength of all contacts located on the equivalent cable within electrotonic distance x of its point of contact with the parent structure, then $F(0) = 0$ and $F(L) = 1$. The function $F(x)$ is now interpolated by the finite Chebyshev series

$$\widehat{F}(x) = \frac{x}{L} + \frac{x(L-x)}{L^2}\sum_{n=0}^{N} a_n\, T_n\Big(\frac{2x-L}{L}\Big) \tag{4.129}$$

where the Chebyshev polynomial $T_n(z)$ is defined by the generating formula $T_n(\cos\theta) = \cos n\theta$. Since expression (4.129) satisfies automatically the conditions $\widehat{F}(0) = 0$ and $\widehat{F}(L) = 1$ for all choices of the coefficients a_n, it remains to find values for $a_0, a_1, \ldots, a_N$ to optimise the fit of expression (4.129) to the cumulative strength of contacts along the entire length of the equivalent cable, as expressed by the set of values $(x_1, F_1), \ldots, (x_M, F_M)$,

where $x_1, \ldots, x_M$ are the points on the equivalent cable. The cumulative strengths of contacts are known and $F_k = F(x_k)$ denotes the cumulative strength of contacts at x_k. In this thesis, the coefficients $a_0, \ldots, a_N$ are determined by minimising

$$\Phi = \sum_{k=1}^{M} \Big[F(x_k) - \frac{x_k}{L} - \frac{x_k(L-x_k)}{L^2} \sum_{n=0}^{N} a_n \, T_n\Big(\frac{2x_k - L}{L}\Big) \Big]^2 . \tag{4.130}$$

It follows immediately from expression (4.130) that

$$\frac{\partial \Phi}{\partial a_j} = -\frac{2}{L^2} \sum_{k=1}^{M} x_k(L-x_k)\, T_j\Big(\frac{2x_k - L}{L}\Big) \Big[F(x_k) - \frac{x_k}{L} - \frac{x_k(L-x_k)}{L^2} \sum_{n=0}^{N} a_n \, T_n\Big(\frac{2x_k - L}{L}\Big) \Big]. \tag{4.131}$$

The optimal values of the coefficients $a_0, \ldots, a_N$ are those which minimise Φ and therefore it clear from equation (4.131) that the optimal coefficients should be chosen to satisfy

$$\begin{aligned} \sum_{n=0}^{N} a_n \sum_{k=1}^{M} \frac{x_k^2(L-x_k)^2}{L^2} T_n\Big(\frac{2x_k - L}{L}\Big) T_j\Big(\frac{2x_k - L}{L}\Big) \\ = \sum_{k=1}^{M} x_k(L-x_k)\Big(F(x_k) - \frac{x_k}{L} \Big) T_j\Big(\frac{2x_k - L}{L}\Big). \end{aligned} \tag{4.132}$$

This is a set of $(N+1)$ simultaneous equations in $(N+1)$ unknowns. These equations are solved for the coefficients $a_0, \ldots, a_N$ by LU decomposition.

The density of contacts on the equivalent cable is estimated from the definition $\widehat{f}(x) = \widehat{F}'(x)$ to obtain

$$\begin{aligned} \widehat{f}(x) &= \frac{1}{L} + \frac{L-2x}{L^2} \sum_{n=0}^{N} a_n \, T_n\Big(\frac{2x - L}{L}\Big) + \frac{x(L-x)}{L^2} \sum_{n=0}^{N} a_n \, \frac{2}{L} T_n'\Big(\frac{2x - L}{L}\Big) \\ &= \frac{1}{L} + \frac{1}{L} \sum_{n=0}^{N} a_n \Big[\frac{L-2x}{L} T_n\Big(\frac{2x - L}{L}\Big) + \frac{2x(L-x)}{L^2} T_n'\Big(\frac{2x - L}{L}\Big) \Big] \end{aligned} \tag{4.133}$$

Give $x \in [0, L]$, the value of $\widehat{f}(x)$ can be computed by first finding θ such that $\cos\theta = (2x - L)/L$. The density now becomes

$$\begin{aligned} \widehat{f}(x) &= \frac{1}{L} + \frac{1}{L} \sum_{n=0}^{N} a_n \Big[-\cos\theta \cos n\theta + \frac{n}{2} \sin\theta \, \sin n\theta \Big] \\ &= \frac{1}{L} + \frac{1}{4L} \sum_{n=0}^{N} a_n \Big[(n-2)\cos(n-1)\theta - (n+2)\cos(n+1)\theta \Big]. \end{aligned} \tag{4.134}$$

Chapter 5

Building the typical neuron

5.1 Introduction

Since Cajal's classic studies using Golgi stained neurones (Cajal, 1952), neurophysiologists, neuroanatomists and more recently mathematicians have tried to understand, describe and model both dendritic morphology and dendritic behaviour. Despite over a century of research, the role of neuronal morphology in shaping neuronal behaviour remains poorly understood. Although the complex morphology of dendritic trees is assumed to be important in determining the properties of spike trains generated by a neuron (*e.g.*, see, Mainen and Sejnowski, 1996; Mel, 1994) little is known about this process. Furthermore, the role of dendritic morphology in integrating the large number of input spike trains distributed across a dendritic tree also remains poorly understood, although a number of studies have addressed this issue (*e.g.*, see Koch, Poggio and Torre, 1982; Shepherd and Brayton, 1987). The first step towards understanding the role played by dendritic morphology in shaping dendritic behaviour is the quantification of dendritic morphology. The aim of this chapter is to provide a novel approach to the problem of simulating dendritic morphology.

Modern computers now enable one to analyse and simulate neuronal morphology. Nevertheless, extracting morphological data is a complex and laborious task, compounded by the diverse structure of dendritic trees. The process of fixing and sectioning the neuron distorts its morphological features leading to possible artifacts in the data. Diameter and length measurements are prone to further error during the reconstruction process due to the resolving power of the microscope and the reconstruction software (Kaspirzhny, Gogan, Horcholle-Bossavit and Tyč-Dumont, 2002) and the subjective nature of the process itself.

Much of the modern work on neuronal morphology was carried out by Hillman (1979), who introduced seven *fundamental* parameters to describe neuronal shape based on the assumption that the cytoskeleton imposes a lower limit on the cross-sectional area of the dendrite. Hillman's parameters are initial and terminal segment diameter, segment length and taper, the ratio between cross-sectional areas of daughter branches, branch power[1] and the spatial orientation of segments (Hillman, 1979). The size of the dendritic tree is specified by the first five parameters, while the shape requires the specification of all seven parameters. From his studies, Hillman asserted that branch power and daughter branch ratio made the most significant contribution to the overall shape of the dendritic tree.

There are two main approaches to the reconstruction of neuronal morphology based on anatomical measurements. The first approach typically models dendritic growth *in vitro* and is referred to as the ontogenetic method, while the second approach simulates the fully developed structure and is referred to as the phylogenetic method. The modelling of dendritic growth *in vitro* concentrates primarily on the probability of a branch occurring as the dendritic segment increases in length (Uemura, Carriquiry, Kliemann and Goodwin, 1995). This procedure is limited as the neurons are grown in culture, and are not subject to many of the factors that may influence dendritic growth *in vivo*. The phylogenetic approach involves the simulation of mature neurons and proceeds by generating either a one-dimensional or three-dimensional representation of a neuron.

(a) The one-dimensional representation of neurons concentrates primarily on characterising their branching properties, and is subdivided into two categories, namely approaches which focus on diameter and approaches which focus on branch order. Hillman (1979), Burke *et al.* (1992) and Ascoli *et al.* (2001) base their analysis on diameter and use it to determine whether a limb will branch or terminate. On the other hand, Van Pelt & Uylings (1999) and Devaud, Quenet, Gascuel and Masson (2000) primarily base their analysis on a description of the possible branching patterns based on the number of terminal segments. Dendritic diameters and lengths may be included in this description, but this is done once the branching pattern has been determined (Van Pelt & Uylings, 2002).

(b) The modelling of spatially orientated three-dimensional dendrites is not well devel-

[1]The power of a branch point is the ratio of the sum of the 3/2 power of the daughter diameters to the 3/2 power of the parent diameter. In a Rall tree this is unity at each branch point (Rall, 1959).

oped. Only limited progress has been made on the detailed quantification of the parameters required to describe three-dimensional structures (Cullhiem, Fleshman, Glenn and Burke, 1987). Tamori (1993) extended the work of Hillman (1979) by adding the additional parameter *effective volume*, which is used in the calculation of branch angles. Factors which influence the direction of growth (tropism) are introduced within three-dimensional models and have a profound effect on the development of the model dendrite (see Ascoli & Krichmar, 2000; Ascoli *et al.*, 2001).

5.1.1 Some recent models of dendritic morphology

The aim of this section is to review briefly current models that account for dendritic morphology.

Burke, Marks and Ulfhake Burke *et al.* (1992) developed a recursive algorithm based on empirical distributions for length and diameter of dendritic segments, and binary branching. They state that a successful simulation of dendritic morphology requires

> "...a method to produce individual branches that have the correct distributions of diameters and lengths, as well as the correct proportions of branches that either branch again or terminate."

In this model, segment lengths grow by increments ΔL (an arbitrary value not derived from the data) and segment diameters decrease by an amount which depends on the taper rate for that segment. After each increment in length, the segment may continue, terminate or branch depending on a set of probabilistic rules. Variations in dendritic shape are determined by this stochastic process. When segments branch, the daughter diameters are calculated by a process that preserves the observed correlation between daughter branch diameters, given the empirical distributions of the diameter of the daughter branches. Discrepancies between observed and simulated distributions of the number of branch points and number of terminations as functions of distance from the soma led to a revised model (see, Burke *et al.*, 1992, Fig 8C), in which they introduced a "grandparent correction" to the original model. They state that the role of this correction is to incorporate "memory" into the process that generates daughter diameters at a branch point. Although this revision improves the fit between simulated and sampled data this is achieved at the cost of a significantly more complicated model that is more difficult to implement.

Ascoli and Krichmar Research on the simulation of neuronal morphology has culminated in L-NEURON, *'a software package for the generation and study of anatomically accurate neuronal analogs'* (Ascoli & Krichmar, 2000). L-NEURON is based on an algorithm for simulating branching patterns in trees (Lindenmayer, 1968). It provides the structural basis for combining the shape parameters defined by Hillman (1979), the effective volume introduced by Tamori (1993) and the algorithm proposed by Burke *et al.* (1992) into a single program to simulate dendritic morphologies. Further parameters describing the orientation of dendrites and branching angles were added to allow the simulation of three-dimensional dendrites. Finally, Rall's power rule was relaxed by multiplying the parent diameter by a constant factor to reflect the experimental data (Ascoli *et al.*, 2001). L-NEURON uses the experimental data directly and returns a 'character string' with specific drawing commands (*i.e.* grow forward, branch, taper etc.) that can be transformed into graphical images of three-dimensional spatially orientated neurons (Ascoli *et al.*, 2001). L-NEURON also includes the global parameter tropism as a modification after the generation of the cells. However, L-NEURON does not appear to contain any procedure to assess the quality with which properties of the sampled neurons are reflected within the simulated neurons, and in particular, properties that have not been used in the simulation process.

Treatment of taper

Models that attempt to simulate neuronal morphology struggle with vast parameter sets and are further complicated by correction factors when the models fail to capture the properties of the original sample. Taper is probably the most difficult parameter to manage in neuronal simulation. Burke *et al.* (1992) found the simulation of branch length to be particularly sensitive to their initial choice of taper despite basing this choice on the experimentally observed data. In an attempt to rectify this problem, they ran a number of simulations with a range of taper rates, and then calculated the root mean square error (E_{RMS}) of the deviations of the simulated length distributions from observed distributions for each rate. The optimal taper rate was chosen as that which minimised E_{RMS} for all branches. Hillman (1979) does include taper as one of the fundamental parameters, yet makes little reference to it. Tamori (1993) excludes taper from his parameter set and instead uses the averaged diameter along a segment. Ascoli *et al.* (2001) do not comment on taper, but it is assumed that they implement taper within the implementation of L-

NEURON. Models developed to simulate branch order (Van Pelt & Uylings, 1999 and Devaud *et al*, 2000) are not concerned with modelling taper.

5.1.2 A new approach to the simulation of dendritic morphology

The models described above inherit their complexity through the absence of a simple principle underlying the development of dendritic morphology and therefore a new approach is required in the analysis and simulation of neuronal morphology. Toward this end, I introduce a procedure based on a single assumption, namely, that a dendritic section of a given diameter will have the same length distribution independent of its position in the dendritic tree. Given this assumption, the dendritic section is taken to be the basic building block of a dendrite. A recursive algorithm based on a simple set of rules, using probability densities estimated from real data, is developed that will generate a dendrite with statistical properties that are statistically indistinguishable from those of the original sample. The success of the algorithm is demonstrated by showing that the original sample and simulated samples preserve several morphological characteristics that were independent of the simulation procedure.

5.2 Mathematical preliminaries

This section sets out the definitions of the various probability densities used in the simulation of dendritic morphology.

5.2.1 Probability density function

The function $f(x)$ is a probability density function on the interval $[a, b]$ provided $f(x) \geq 0$ for all $x \in [a, b]$ and

$$\int_a^b f(x)dx = 1. \tag{5.1}$$

Probability is associated with area under the probability density function, and so the probability that the random variable $X \in [c, d]$ is

$$\text{Prob}(c \leq X \leq d) = \int_c^d f(x)\, dx$$

where $[c, d] \subseteq [a, b]$.

5.2.2 Joint probability density function

The idea of a probability function in one dimension may be extended to two or more dimensions to give what is often called a joint probability density function. For example, $f(x, y)$ may be interpreted as a joint probability density function of the random pair (X, Y) over $\mathbb{R}^2$ provided $f(x, y) \geq 0$ for all x and y and

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x, y)\, dx\, dy = 1\,. \tag{5.2}$$

The density function may be regarded as a probability surface in three dimensions. Probability is measured by the volume under this surface, and so

$$\text{Prob}(a \leq X \leq b \text{ and } c \leq Y \leq d) = \int_c^d \int_a^b f(x, y)\, dx\, dy$$

is the probability that the random variable (X, Y) lies in the rectangle $[a, b] \times [c, d]$.

Marginal densities

Given a joint probability function $f(x, y)$, the density of the random variable X in the absence of information regarding the value of Y is called the *marginal density* of X.

Similarly, the marginal density of Y is the density of Y in the absence of information on the value of X. I shall use the notation

$$\phi_X(x) = \int_{-\infty}^{\infty} f(x,y)\,dy\,, \qquad \phi_Y(y) = \int_{-\infty}^{\infty} f(x,y)\,dx \tag{5.3}$$

to denote the marginal densities of X and Y respectively.

Conditional density

Given a joint probability function $f(x,y)$, the density of the random variable X given that $Y = y$ is called the *conditional density* of X. Similarly, the conditional density of Y is the density of Y given that $X = x$. I shall use the notation

$$f(x \mid Y = y) = \frac{f(x,y)}{\phi_Y(y)}\,, \qquad f(y \mid X = x) = \frac{f(x,y)}{\phi_X(x)} \tag{5.4}$$

to denote respectively the conditional density of X given Y and the conditional density of Y given X. In particular, the deviates X and Y are independent provided the conditional density of X given Y is the marginal density of X, that is, the density of X is entirely independent of Y. In this case the joint probability density function of X and Y is the product of the probability density functions of X and Y.

5.3 A procedure for simulating a sample of dendrites

The aim of this section is to give an overview of the procedure used to simulate the morphology of a typical dendrite from a sample of neurons which are assumed *a priori* to represent a single type of neuron. The work of this chapter distinguishes between interneurons that receive different classes of input, although they may come from the same group of interneurons. The sampled neurons are specified in terms of their dendritic diameter, the coordinates of the points at which the diameters are measured, and information on the pattern of connectivity for each segment. From this data the length of each dendritic segment can be determined, as well as the pattern of connectivity between segments. This work focuses on the development of a procedure to generate a *typical neuron* from a large sample of neurons of a single type.

The procedure to be used in this simulation is motivated by the observation that the dendritic segments in the sample of interneurons at my disposal are largely composed of uniform cylinders, and that changes in segment diameter predominantly occur at branch points or are the result of local discontinuities in diameter along the segment. Based on this observation, dendritic segments will be generated as a sequence of uniform cylinders (sections). The basic assumption of the simulation procedure is that *the combined properties of a dendritic section, namely its diameter and length, are independent of its location in the dendritic tree.* This assumption is the basis of a recursive algorithm that is used to generate model neurons. The operation of the algorithm draws from a series of probability densities that in turn have to be estimated from the sample of neurons. Both the estimation of non-parametric probability densities and the procedures for drawing samples from these densities form an important part of this algorithm. The simulation of a model neuron begins by determining the number of dendrites connected to its soma. This number is obtained by drawing from the distribution of the number of dendrites per neuron in the sample. Once the number of dendrites is selected, the recursive procedure is used to generate the complete structure of each dendrite.

Select stem diameter The diameter of the first stem section is obtained by making a random draw from the estimated distribution of diameters of first stem sections. This procedure is implemented once for each dendrite. Once the diameter of the first stem section is determined, the process continues by following the protocol for generating a dendritic segment as a sequence of dendritic sections.

Generate a dendritic segment Given a diameter d, the length of the associated section is determined by a random draw from the joint distribution of section lengths and section diameters conditioned on the value of d. Once the section is defined, there are three possible continuations; the section terminates, the section continues or the section branches (binary). The probability of each of these events, conditioned on the section diameter, requires estimates for the distribution of the diameters of terminating sections, the distribution of the diameters of continuing sections and the distribution of the diameters of branching sections.

(a) **The section terminates.** The process for generating sections now continues from the most recent incomplete branch point with a known diameter. Figure 5.1 illustrates one possible path in the construction of the dendrite. When segment 3, for example, is complete, the process returns to branch point P_2 and proceeds to construct segment 4. Once segment 4 is complete the process returns to branch point P_1 and constructs segment 5, and so on.

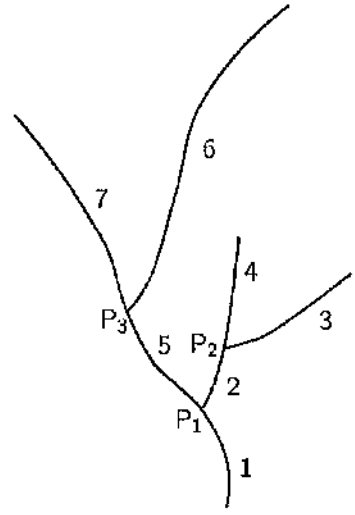

Figure 5.1: An idealised neuron illustrating a possible path of the recursive procedure, segments 1 to 7 and branch points P_1 to P_3.

(b) **The section continues.** The diameter of the next section is determined by a random draw from the joint distribution of the diameters of contiguous sections conditioned on the diameter of the current section. The next segment length is then generated by drawing from the joint distribution of section lengths and section diameters conditioned on the diameter of the new section.

(c) **The section branches.** The joint distribution of the parent diameter and first daughter diameter (defined to be the daughter with larger diameter) is estimated from the sample. The diameter of the first daughter is determined from this density conditioned by the value of the parent diameter. The *trivariate* density of the parent and two daughter diameters is constructed, and the diameter of the second daughter is drawn from this distribution conditioned on the parent and first daughter diameters. The algorithm now follows the path of the first daughter until all branches have terminated before returning to the branch point to follow the path of the second daughter branch, as illustrated in Figure 5.1.

To summarise, the underlying assumption that the section diameter and section length are independent of location within the dendritic tree is the basis for the recursive algorithm for simulating dendritic structure. The implementation of this simulation algorithm requires the construction of various univariate and multivariate probability density functions from the sampled neurons. This is achieved using a non-parametric technique based on kernel density estimation. The technique is described in the following section.

5.4 Estimating distributions

Previous attempts to describe the statistical properties of dendritic morphology have assumed that various features of the morphology follow parametric distributions. For example, Ascoli *et al.* (2001) used Gaussian, gamma or exponential distributions, while Hillman (1979) and Burke *et al.* (1992) modelled morphological data using Gaussian distributions, although the former recognised that this was an inappropriate choice of distribution for the diameter of terminal segments. By contrast to previous studies, this work uses the kernel density estimation technique to characterise the statistical properties of neuronal morphology. The technique generalises the notion of a histogram and leads to a non-parametric estimate of probability density in which each observation in a sample is treated as an independent random variable.

Let $X_1, X_2, \ldots, X_n$ be a sample of n observations with underlying density $f(x)$. The kernel estimate $\widehat{f}(x)$ of $f(x)$ is a representation of the density of X in the form

$$\widehat{f}(x) = \frac{1}{nh} \sum_{k=1}^{n} K\Big(\frac{x - X_k}{h}\Big) \tag{5.5}$$

where $K(x)$ is a non-negative (kernel) function of x satisfying

$$\int_{-\infty}^{\infty} K(x)\, dx = 1. \tag{5.6}$$

The parameter h appearing in formula (5.5) is called the window width, or bandwidth, of the estimator. It follows immediately from the properties of $K(x)$ that $\widehat{f}(x)$ is a probability density, that is, $\widehat{f}(x) \geq 0$ and

$$\int_{-\infty}^{\infty} \widehat{f}(x)\, dx = 1.$$

It turns out in practice to be the value assigned to the bandwidth h that is critical to how well the unknown probability density is estimated by $\widehat{f}(x)$ (see Table 3.1 in Silverman, 1986), and not the choice of the kernel function $K(x)$. Following Silverman (1986), the quality with which $\widehat{f}(x)$ estimates $f(x)$ is measured by the Mean Integrated Square Error (MISE) defined by

$$\mathrm{MISE}\,(\widehat{f}\,) = \mathrm{E}\Big[\int_{-\infty}^{\infty} \big(\widehat{f}(x) - f(x)\big)^2\, dx\Big]. \tag{5.7}$$

Let $\tilde{f}(x) = \mathrm{E}\,[\,\widehat{f}(x)\,]$. The result of taking the expected value operator inside the integral

is

$$\begin{aligned}\mathrm{MISE}(\widehat{f}\,) &= \int_{-\infty}^{\infty} \mathrm{E}\left[\,(\,\widehat{f}(x) - f(x)\,)^2\,\right] dx \\ &= \int_{-\infty}^{\infty} \mathrm{E}\left[\,(\,\widehat{f}(x) - \bar{f}(x)\,)^2\,\right] dx + \int_{-\infty}^{\infty} (\,\bar{f}(x) - f(x)\,)^2\, dx \qquad (5.8) \\ &\quad +2\int_{-\infty}^{\infty} \mathrm{E}\left[\,(\,\widehat{f}(x) - \bar{f}(x)\,)(\,\bar{f}(x) - f(x)\,)\,\right] dx\,.\end{aligned}$$

The first integral on the right hand side of equation (5.8) is the integrated variance of the operator. The second integral on the right hand side of equation (5.8) is the integrated squared bias[2] of the MISE operator, and the third integral is zero by the definition of $\bar{f}(x)$. Thus the MISE can be simplified to give

$$\mathrm{MISE}(\widehat{f}\,) = \int_{-\infty}^{\infty} \mathrm{E}\left[\,(\,\widehat{f}(x) - \bar{f}(x)\,)^2\,\right] dx + \int_{-\infty}^{\infty} (\,\bar{f}(x) - f(x)\,)^2\, dx\,. \qquad (5.9)$$

The kernel bandwidth h is chosen to minimise the MISE. Silverman (1986) derives approximate expressions for both components of the MISE defined in equation (5.9) for kernel functions $K(x)$ satisfying the conditions

$$\int_{-\infty}^{\infty} K(t)\, dt = 1, \qquad \int_{-\infty}^{\infty} t\, K(t)\, dt = 0\,, \qquad \int_{-\infty}^{\infty} t^2 K(t)\, dt = \mu_2. \qquad (5.10)$$

With these assumptions Silverman (1986) shows that the MISE is well approximated by the expression

$$\mathrm{MISE}(\widehat{f}\,) \approx \frac{R(K)}{nh} + \frac{h^4 \mu_2^2 R(f'')}{4}\,, \qquad R(\psi) = \int_{-\infty}^{\infty} \psi^2(x)\, dx \qquad (5.11)$$

where $f''(x)$ is the second derivative of the probability density function, and the first and second terms on the right-hand side of (5.11) are respectively the integrated variance and integrated square bias of the MISE operator. Clearly small bandwidths reduce bias in the estimate $\widehat{f}(x)$, but at the cost of increasing its variance. On the other hand large bandwidths reduce the variance of the estimate, but at the expense of increasing its bias. An expression for the optimal bandwidth can be found by differentiating expression (5.11) with respect to h and finding the value of h for which this derivative is zero. It follows from expression (5.11) that

$$\frac{d\mathrm{MISE}}{dh} = -\frac{R(K)}{nh^2} + h^3 \mu_2^2 R(f'')\,, \qquad (5.12)$$

[2]Bias measures the difference between the expected value of the estimator and the actual value of the quantity being estimated - in this case $\bar{f}(x) - f(x)$.

and this is in turn zero when

$$h = \Big[\frac{R(K)}{n\mu_2^2 R(f'')} \Big]^{1/5}. \tag{5.13}$$

The difficulty with expression (5.13) is that the optimal bandwidth h is expressed in terms of $R(f'')$, the second order roughness of the true density, which is unknown. However, if the density appears to be lumped then $R(f'')$ can often be estimated by first fitting the observations to a Gaussian distribution, and replacing $R(f'')$ by its value computed analytically from the Gaussian distribution with parameters calculated from the data. Furthermore, if $K(x)$ is taken to be the probability density for the N(0,1) distribution, then it is a matter of straight forward Calculus to show that the optimal bandwidth has value

$$h = 1.06\,\sigma\, n^{-1/5} \tag{5.14}$$

where σ is the standard deviation of the observations. The final expression for the kernel estimate $\widehat{f}(x)$ of the true density $f(x)$ in the case when $K(x)$ is taken to be the probability density for the N(0,1) distribution is

$$\widehat{f}(x) = \frac{1}{nh\sqrt{2\pi}} \sum_{k=1}^{n} \exp\Big[-\frac{1}{2}\Big(\frac{x - X_k}{h}\Big)^2 \Big]. \tag{5.15}$$

5.4.1 Multivariate kernel estimates of density

The univariate kernel estimation procedure just described for one dimension can be generalised to the estimation of joint probability density functions in two and three dimensions. For example, multivariate estimates of probability density are required in the analysis of dendritic branch points.

Bivariate density

Suppose that $(X_1, Y_1), \ldots, (X_n, Y_n)$ are n bivariate observations, then the kernel density estimate $\widehat{f}(x, y)$ of the joint probability density function $f(x, y)$ is

$$\widehat{f}\,(x, y) = \frac{1}{nh_x h_y} \sum_{k=1}^{n} K\Big(\frac{x - X_k}{h_x}\Big) K\Big(\frac{y - Y_k}{h_y}\Big) \tag{5.16}$$

where h_x and h_y are the bandwidths for X and Y respectively. For example, X might denote the diameter of a dendritic section and Y might denote its corresponding length.

Trivariate density

Suppose that $(X_1, Y_1, Z_1), \ldots, (X_n, Y_n, Z_n)$ are n trivariate observations then the kernel density estimate $\widehat{f}(x, y, z)$ of the joint probability density function $f(x, y, z)$ is

$$\widehat{f}(x, y, z) = \frac{1}{n h_x h_y h_z} \sum_{k=1}^{n} K\Big(\frac{x - X_k}{h_x}\Big) K\Big(\frac{y - Y_k}{h_y}\Big) K\Big(\frac{z - Z_k}{h_z}\Big) \tag{5.17}$$

where h_x, h_y and h_z are the bandwidths for X, Y and Z. For example, X might denote the parent diameter of a dendritic section at a dendritic branch point and Y and Z might denote the two daughter diameters at that branch point.

5.4.2 Marginal and conditional densities

Two marginal densities are required in the construction procedure. For the bivariate distribution (5.16), the marginal density $\widehat{\phi}_X$ of X and $\widehat{\phi}_Y$ of Y are derived from equation (5.3) and have respective values

$$\widehat{\phi}_X(x) = \frac{1}{n h_x} \sum_{k=1}^{n} K\Big(\frac{x - X_k}{h_x}\Big), \qquad \widehat{\phi}_Y(y) = \frac{1}{n h_y} \sum_{k=1}^{n} K\Big(\frac{y - Y_k}{h_y}\Big) \tag{5.18}$$

where the kernel properties of $K(x)$ have been used in the calculation of these probability densities.

Conditional distributions are used frequently in the simulation of dendrites. For example, the properties of dendritic sections are controlled by the joint distribution of section lengths and diameters. Once a diameter of a section is known, its length must be drawn from the joint distribution of diameters and lengths conditioned on that value of diameter. If X denotes section length and Y denotes section diameter, then the kernel estimate of the conditional probability of X in the joint density (5.16), given that $Y = y$, is

$$\begin{aligned} \widehat{f}(x \mid Y = y) &= \frac{\dfrac{1}{n h_x h_y} \displaystyle\sum_{k=1}^{n} K\Big(\frac{x - X_k}{h_x}\Big) K\Big(\frac{y - Y_k}{h_y}\Big)}{\displaystyle\int_{-\infty}^{\infty} \frac{1}{n h_x h_y} \sum_{k=1}^{n} K\Big(\frac{x - X_k}{h_x}\Big) K\Big(\frac{y - Y_k}{h_y}\Big)\, dx}, \\ &= \frac{\dfrac{1}{h_x} \displaystyle\sum_{k=1}^{n} K\Big(\frac{x - X_k}{h_x}\Big) K\Big(\frac{y - Y_k}{h_y}\Big)}{\displaystyle\sum_{k=1}^{n} K\Big(\frac{y - Y_k}{h_y}\Big)}. \end{aligned} \tag{5.19}$$

Clearly a similar expression exists for the distribution of Y conditioned on $X = x$. However, in the example just described, section diameter is always known and it is section

length that has to be found, and therefore the density of section diameters conditioned on section length is never needed.

As a further example of the occurrence of a conditional distribution, suppose that the sample observations are n triples of binary branch point diameters in which X denotes the parent diameter, Y denotes the first daughter diameter and Z denotes the second daughter diameter. The observations are used to construct the joint probability density function of (X, Y, Z), and from this density it is required to draw a second daughter diameter given the diameters of the parent and first daughter sections. To construct the conditional joint distribution function of the diameter of a second daughter section, given a parent section with diameter $X = x$ and a first daughter section with diameter $Y = y$, it is first necessary to compute the kernel estimate of the marginal distribution of the diameters of parent and first daughter sections from the joint distribution function (5.17). This marginal density is

$$\begin{aligned}\widehat{\phi}_{XY}(x,y) &= \frac{1}{nh_xh_yh_z}\sum_{k=1}^{n}\int_{-\infty}^{\infty} K\Big(\frac{x-X_k}{h_x}\Big)K\Big(\frac{y-Y_k}{h_y}\Big)K\Big(\frac{z-Z_k}{h_z}\Big)\,dz \\ &= \frac{1}{nh_xh_y}\sum_{k=1}^{n} K\Big(\frac{x-X_k}{h_x}\Big)K\Big(\frac{y-Y_k}{h_y}\Big),\end{aligned}$$

and this is now used to construct the conditional joint distribution function of the diameter of second daughter sections, conditioned on a parent section with diameter $X = x$ and a first daughter section with diameter $Y = y$. The result is

$$\widehat{f}(z \mid X = x,\ Y = y) = \frac{\dfrac{1}{nh_xh_yh_z}\displaystyle\sum_{k=1}^{n} K\Big(\frac{x-X_k}{h_x}\Big)K\Big(\frac{y-Y_k}{h_y}\Big)K\Big(\frac{z-Z_k}{h_z}\Big)}{\widehat{\phi}_{XY}(x,y)}. \qquad (5.20)$$

5.4.3 Bandwidth selection

The choice of bandwidths to be used in the kernel estimation of univariate and multivariate distributions are chosen with reference to the Gaussian distribution as advocated by Silverman (1986). This reference distribution is used to estimate the various roughness properties (*e.g.*, $R(f'')$) of the true distribution that may be required in the estimation of optimal bandwidths in the kernel estimate of probability density. Furthermore, $K(x)$ will be assumed to be the distribution function for an N(0,1) deviate, that is, $K(x) = (2\pi)^{-1/2}\,e^{-x^2/2}$. With these assumptions, it has already been shown that the optimal bandwidth for univariate kernel density estimation is $h = 1.06\,\sigma\,n^{-1/5}$, where σ

is the standard deviation of the sample data, and n is the number of observations in the sample (Silverman, 1986).

Bandwidths in higher dimensions

In the case of multivariate distributions, the choice of bandwidths must take into account correlations between variates. For a bivariate distribution, the bandwidths are calculated using the formula

$$\begin{aligned} h_x &= \sigma_x(1-\rho^2)^{5/12}(1+\rho^2/2)^{-1/6}n^{-1/6}, \\ h_y &= \sigma_y(1-\rho^2)^{5/12}(1+\rho^2/2)^{-1/6}n^{-1/6} \end{aligned} \tag{5.21}$$

where σ_x and σ_y are respectively the standard deviations of X and Y and $\rho \in [-1,1]$ is the correlation coefficient, defined as

$$\rho = \frac{\mathrm{E}\left[\left(X-\mathrm{E}[X]\right)\left(Y-\mathrm{E}[Y]\right)\right]}{\sigma_x\sigma_y} = \frac{\sigma_{xy}}{\sigma_x\,\sigma_y}, \tag{5.22}$$

in which σ_{xy} is the sample covariance. Expressions (5.21) for the bandwidths h_x and h_y allow for correlation between the deviates X and Y. If $\rho^2 = 0$, then equations (5.21) reduce to the equations given by Silverman (1986, Table 4.1) for multivariate bandwidth selection.

Following Silverman (1986), the "normal reference rule" for bandwidth selection in a sample space of dimension d is

$$h_k = \sigma_k \left[\frac{4}{d+2}\right]^{1/(d+4)} n^{-1/(d+4)} \tag{5.23}$$

where h_k $(1 \leq k \leq d)$ is the bandwidth for the k-th deviate, and σ_k is its standard deviation. The coefficient $\left[\frac{4}{d+2}\right]^{1/(d+4)}$ ranges from unity when $d = 2$ to a minimum value of approximately 0.924 when $d = 11$, and thereafter increases monotonically to its limiting value of unity as d becomes arbitrarily large. Scott (1992) suggests that the sensitivity of the coefficient $\left[\frac{4}{d+2}\right]^{1/(d+4)}$ to changes in d is sufficiently small that this coefficient can be treated as unity without detriment. Scott therefore suggests the use of the simplified formula

$$h_k = \sigma_k\, n^{-1/(d+4)}. \tag{5.24}$$

5.4.4 Comparison of distributions

Kernel density estimation allows the comparison of two distributions in one or more dimensions. The statistic used in this work as the basis of the comparison is the inte-

grated squared difference of probability density functions. Suppose that f_1 and f_2 are two probability density functions defined over a sample space $\mathcal{D}$, then the integrated squared difference of f_1 and f_2 is

$$\Psi = \int (f_1 - f_2)^2 \, d\mathcal{D} \tag{5.25}$$

where the integration is taken over the sample space $\mathcal{D}$. Clearly if $f_1 \equiv f_2$ then $\Psi \equiv 0$. Departures from $\Psi = 0$ measure the extent to which f_1 is different from f_2.

Suppose now that f_1 and f_2 are kernel estimates of probability density based on samples $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively. The problem is to determine whether or not the sample $\mathcal{S}_1$ is different from the sample $\mathcal{S}_2$. Although, of course, there is no way of giving a definitive answer to this question, the statistic defined in (5.25) can be used as the basis for a hypothesis test which will provide a probability that the null hypothesis is true, namely that $\mathcal{S}_1$ and $\mathcal{S}_2$ are samples drawn from the same distribution.

The value of Ψ is first computed for the samples $\mathcal{S}_1$ and $\mathcal{S}_2$ with f_1 and f_2 replaced by their kernel density estimates $\widehat{f_1}$ and $\widehat{f_2}$ respectively. On the basis of the null hypothesis, namely that samples $\mathcal{S}_1$ and $\mathcal{S}_2$ are distributed identically, the data from both samples are pooled. A simulated pair of data sets is now constructed from the pooled data without replacement, and the value for Ψ calculated for this simulated pair. By repeating the procedure of drawing random pairs of sample sets from the pooled data, the distribution of the statistic Ψ is computed on the basis that the samples $\mathcal{S}_1$ and $\mathcal{S}_2$ are distributed identically. A univariate kernel density estimate of the distribution of Ψ is now constructed from the simulated values of Ψ, and the position of the real value of Ψ compared against this distribution which was constructed under the null hypothesis that $\mathcal{S}_1$ and $\mathcal{S}_2$ are distributed identically. If the actual value of Ψ lies in the tails of the kernel estimate of the distribution of Ψ, namely the first or last 2.5%, then the samples $\mathcal{S}_1$ and $\mathcal{S}_2$ are deemed to be distributed differently at the 5% level of significance.

A test of the comparison procedure

The aim of this section is to test the validity of the procedure used to compare the difference between two non-parametric distributions. Recall that a Type 1 error is the probability of rejecting the null hypothesis when it is true. If the significance of the hypothesis test is set at p, then fraction p of test statistics must fail the test when the hypothesis is true. This statement is valid for all values of p, for example, $p = 0.01$ and $p = 0.05$ are common

choices for p, and therefore the test statistic must be uniformly distributed in $[0,1]$. To test this property of Ψ, two samples $\mathcal{S}_1$ and $\mathcal{S}_2$ are constructed from a normal distribution with mean μ and standard deviation σ. The actual value of the statistic Ψ is computed, and then the samples are pooled and 200 simulations of Ψ are constructed by drawing from the pooled data without replacement. The probability of obtaining the actual statistic is then estimated. Figure 5.2 shows the distribution of 2000 repetitions of this experiment. Values of Ψ are binned at intervals of 0.05 on the horizontal axis and the count is displayed on the vertical axis.

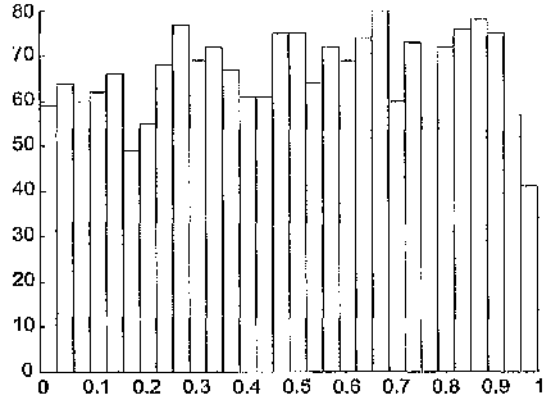

Figure 5.2: The histogram shows the result of 2000 repetitions of the probability with which the actual statistic is realised within 200 simulated pairs of samples drawn from the pooled data on the basis of the null hypothesis.

It is clear from this figure that the probability returned by any particular simulation is uniformly distributed in $[0,1]$. Therefore if the significance level is set at $p\%$, then there is a $p\%$ likelihood of rejecting the hypothesis when it is true.

5.4.5 Drawing from a random distribution

On many occasions it is required to draw a random variable from a kernel estimate of density based on a sample of n observations. The first stage of the process is to use a uniform random number generator to choose an integer from 1 to n. This is most effectively done by calculating $h = 1/n$ and then determining the random integer between 1 and n by the formula $k = \lfloor U(0,1)/h \rfloor + 1$ where $U(0,1)$ is a uniformly distributed random number in $(0,1)$ and $\lfloor \psi \rfloor$ denotes the integer part of ψ.

Once k is determined, the deviate X_k identifies the observation about which the drawing is to be made. Another uniformly distributed random number U is drawn and leads to

the random number $X_k + h\phi^{-1}(U)$ where $\phi(x)$ is the cumulative distribution function of the Gaussian distribution with zero mean and unit standard deviation. Since $y = \phi^{-1}(U)$ is not available as a standard function (only $\phi(x)$ is available), the usual way to find y is to use the bisection algorithm to solve the equation $U = \phi(y)$.

5.5 Validation of the model assumption

The assumption underlying the recursive algorithm for simulating dendritic morphology is that the diameter and length of a section is independent of its location within the dendritic tree. The aim of this section is to validate this assumption by examining the distributions of dendritic sections from stem, intermediate and terminal segments with respect to length over selected diameter ranges.

Table 5.1 illustrates the diameter ranges, the number of sections in each range and the p-value associated with the test that the distribution of lengths within these ranges are identical.

Diameter	No. of sections in dendritic regions			p-value: hypothesis
Range (μm)	Stem	Intermediate	Terminal	of identical lengths
$1.00 < d < 1.50$		137	228	0.983
$1.50 < d < 2.00$		154	110	0.989
$1.50 < d < 2.50$		212	135	0.681
$1.50 < d < 3.50$		243	147	0.436
$2.00 < d < 3.00$	113		36	0.942
$2.50 < d < 3.50$	60		10	0.726*
$2.25 < d < 2.75$	49	25		0.755*
$2.75 < d < 3.25$	37	13		0.168*
$2.25 < d < 3.25$	86	38		0.294
$2.25 < d < 3.75$	105	43		0.484

Table 5.1: Comparison of section lengths for interneurons receiving myelinated input partitioned by section diameter. The left column of the table gives the diameter range over which the distributions of lengths are compared, the middle column gives the number of sections for each dendritic region in the diameter range, and the right column gives the p-values for the hypothesis that the distribution of lengths within this range are identical. The distributions were compared using the procedure described in Subsection 5.4.4, except for those marked * where the data is discrete and a two-sample t-test was used.

The table is based on thirty-one cholinergic interneurons receiving myelinated input. These neurons gave rise to 272 stem sections, 428 intermediate sections and 577 terminal sections. Unbranched dendrites were omitted from this analysis as they would contribute to both the

stem and terminal categories and would therefore bias the result[3]. In total, 40 sections were omitted. Similarly, twenty cholinergic interneurons receiving unmyelinated input gave 240 stem sections, 349 intermediate sections and 556 terminal sections. Table 5.2 illustrates the diameter ranges, the number of sections in each range and the p-value associated with the test that the distribution of lengths within these ranges are identical. Again, unbranched dendrites were omitted from the analysis, 23 sections in this case.

Diameter Range (μm)	No. of sections in dendritic regions			p-value: hypothesis of identical lengths
	Stem	Intermediate	Terminal	
$1.00 < d < 2.00$		120	159	0.872
$2.00 < d < 3.00$		81	48	0.469
$2.00 < d < 4.00$		129	63	0.392
$2.50 < d < 3.50$	53		24	0.711
$2.50 < d < 4.00$	97		29	0.436
$2.00 < d < 3.00$	47	81		0.053
$3.00 < d < 4.00$	74	48		0.665
$2.00 < d < 4.00$	121	129		0.488
$2.50 < d < 4.50$	119	91		0.356
$3.00 < d < 5.00$	116	65		0.962

Table 5.2: Comparison of section lengths for interneurons receiving unmyelinated input partitioned by section diameter. The left column of the table gives the diameter range over which the distributions of lengths are compared, the middle column gives the number of sections for each dendritic region in the diameter range, and the right column gives the p-values for the hypothesis that the distribution of lengths within this range are identical. The distributions were compared using the procedure described in Subsection 5.4.4.

It is clear from Tables 5.1 and 5.2 that the distributions of lengths within each diameter range are statistically indistinguishable. For example, consider interneurons receiving myelinated input (Table 5.1) for diameters lying in the range 1.0 to 1.5 μm. The p-value for a comparison of the distributions of lengths for intermediate and terminal sections is $p = 0.938$. A similarly strong result holds for each diameter range for both classes of interneuron. Now that the underlying assumption for the procedure for constructing a typical dendrite has been validated, it remains to describe this procedure.

[3]Although these sections are removed when testing the validity of the assumption since they introduce bias, the construction procedure generates unbranched dendrites, and their number and structure can be compared with those observed in the sample to provide a second and independent test of the model.

5.6 Formal procedure for constructing a dendrite

The formal procedure for simulating neuronal morphology is illustrated for interneurons receiving myelinated afferent input. The same procedure can be applied to interneurons receiving unmyelinated afferent input.

The first step in the construction procedure is to select at random the number of dendrites that are to be constructed for the interneuron. Figure 5.3A (clear bars) shows the histogram for the number of dendrites per cell for interneurons receiving myelinated afferent input (Figure 5.3A, solid bars, refers to interneurons receiving unmyelinated afferent input). Once the number of dendrites is selected, the recursive procedure outlined in Section 5.3 begins.

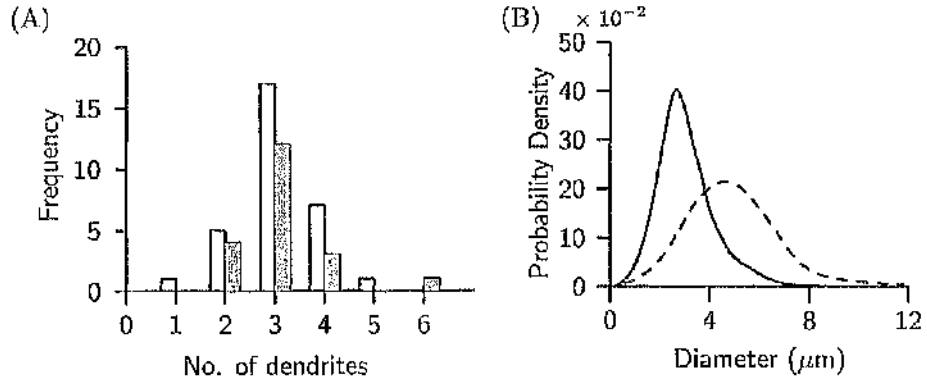

Figure 5.3: (A) Histogram shows the frequency of dendrites per cell for interneurons receiving myelinated input (clear) and unmyelinated input (shaded). (B) Distribution of stem section diameters for neurons receiving myelinated (solid line) and unmyelinated (dashed line) input.

The recursive algorithm to construct a dendrite starts with the determination of the diameter of the first stem section. Figure 5.3B (solid line) illustrates the distribution of the diameters of first stem sections for interneurons receiving myelinated input (Figure 5.3B, dashed line, applies to neurons receiving unmyelinated input). The diameter of the first stem section is drawn from this distribution.

Once this diameter is known, the length of the associated section must be determined. The latter, involves the drawing of a sample at random from the joint distribution of all section lengths and diameters conditioned by the section diameter. This distribution is illustrated in Figure 5.4A,C.

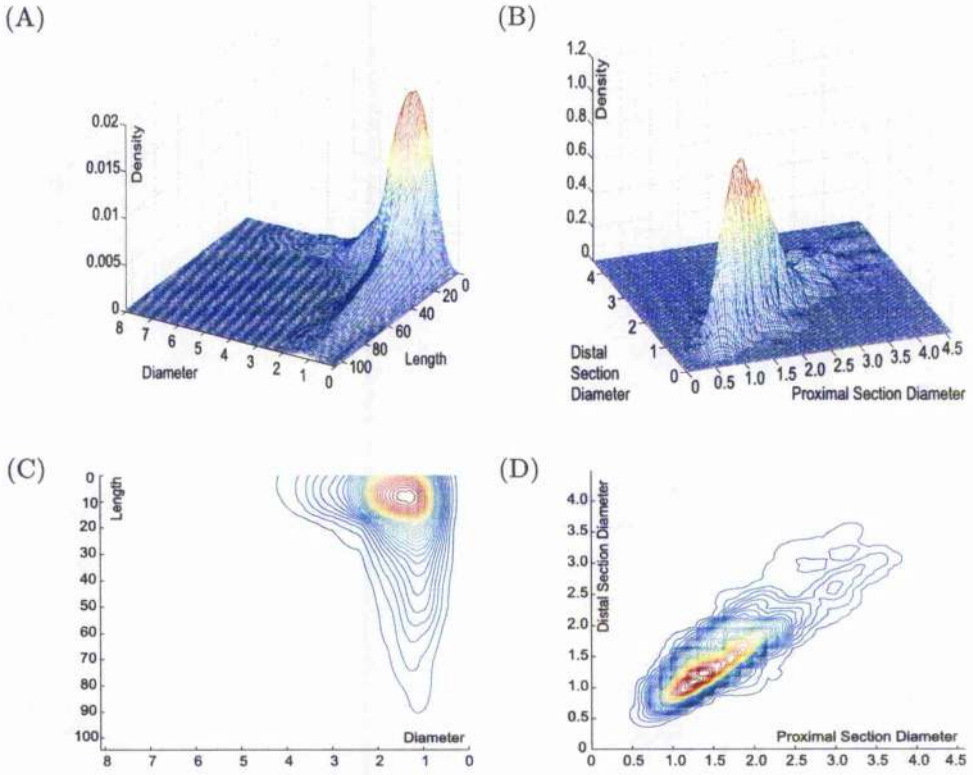

Figure 5.4: Estimated bivariate densities for (A) and (C) all section diameters and all section lengths, and (B) and (D) all pairs of contiguous sections from interneurons receiving myelinated input (referred to as proximal and distal sections). (A) and (C) show the estimated density and contour plot for all section diameters and all section lengths, and (B) and (D) show the estimated density and contour plot for all contiguous sections. All lengths and diameters are measured in μm.

Once the length of the section has been found, it is necessary to determine if it terminates, continues or branches. This selection process requires the estimation of the probability density of the diameter of sections that terminate, the diameter of sections that continue and the diameter of sections that branch. These densities for interneurons receiving myelinated input are shown in Figure 5.5A (Figure 5.5B shows the distribution for unmyelinated input).

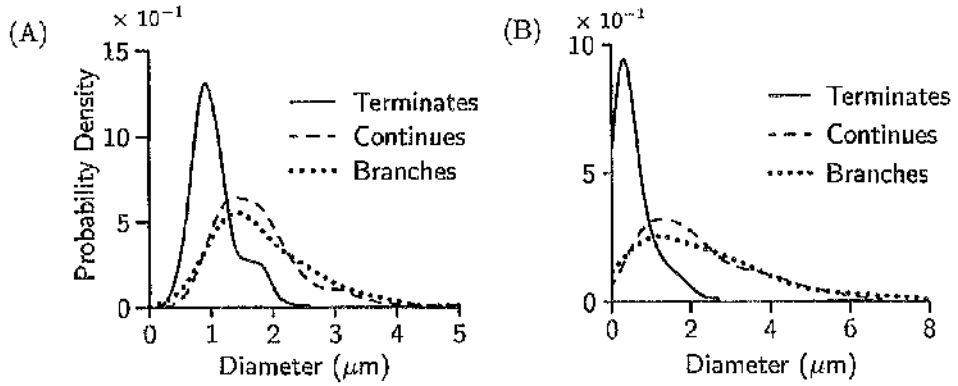

Figure 5.5: Kernel density estimates for the distributions of diameters from sections that terminate (solid), continue (dashed) or branch (dotted) for interneurons receiving myelinated input (A) and unmyelinated input (B).

For a given section diameter d, let $f_t(d)$, $f_c(d)$ and $f_b(d)$ be the respective values of the probability density function of terminating sections, continuing sections and branching sections at diameter d, then the probabilities of terminating, continuing and branching are respectively

$$\begin{aligned} P_t &= \frac{n_t f_t(d)}{n_t f_t(d) + n_c f_c(d) + n_b f_b(d)} \\ P_c &= \frac{n_c f_c(d)}{n_t f_t(d) + n_c f_c(d) + n_b f_b(d)} \\ P_b &= \frac{n_b f_b(d)}{n_t f_t(d) + n_c f_c(d) + n_b f_b(d)}. \end{aligned} \tag{5.26}$$

where n_t, n_c and n_b are the number of observations in the samples of terminating, continuing and branching sections. The interval $[0, 1]$ is subdivided into three subintervals $[0, P_t]$, $[P_t, P_t + P_c]$ and $[P_t + P_c, 1]$ and a uniform random number in $[0, 1]$ is drawn. The interval containing this value determines if the section terminates, branches or continues.

When section termination is selected the construction procedure returns to the most recent branch point and repeats the process of determining the characteristics of the subsequent sections (see Figure 5.1).

If section continuation is selected, then the diameter of the next section is drawn from the estimated joint distribution of contiguous sections (see Figure 5.4B,D), conditioned by the diameter of the current section. Once the diameter of the section is determined, the construction process repeats the standard procedure for finding a section length, and

determining the end condition of that section, namely, terminate, continue or branch.

If section branching is selected, daughter sections must be constructed. For this purpose it is convenient to designate the daughter section with the larger diameter as the first daughter section. The relationship between diameters of the parent section and first daughter section at a branch point is illustrated in Figure 5.6A,B. It is clear from this figure that these deviates are strongly correlated and this correlation has been recognised in the construction of the bandwidths on which the kernel estimate of probability density is based. The diameter of the first daughter section is now drawn from the joint distribution of parent

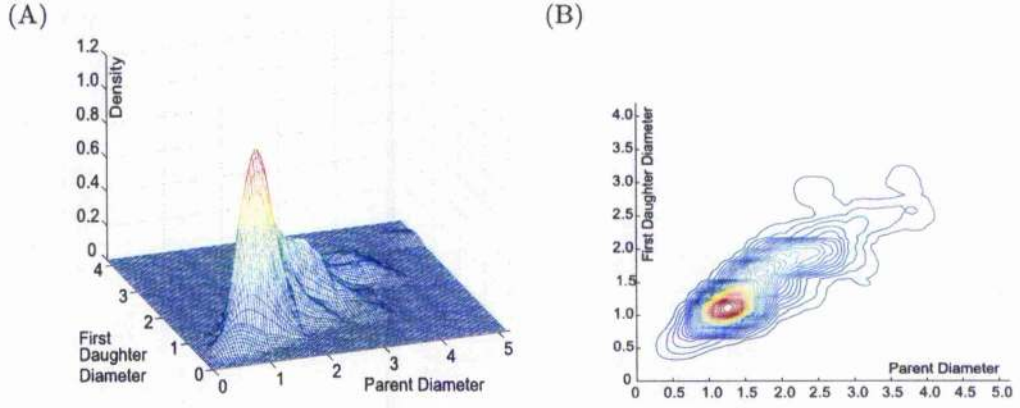

Figure 5.6: Branch point distributions from interneurons receiving myelinated input. (A) The joint distribution of parent section and first daughter diameters and (B) the corresponding contour plot. All values are measured in μm.

and first daughter diameters, conditioned by the (known) diameter of the parent section. Once the parent diameter and the diameter of the first daughter section are determined, the diameter of the second daughter section is found directly from the trivariate density of the parent, first daughter and second daughter, conditioned on the (known) diameters of the parent and first daughter sections. Thereafter the length of each daughter section is determined by the standard procedure.

At a branch point, the construction process generates the diameters of the two daughter sections, then using the first daughter section it follows that path, adding sections and branching until a section terminates. After a termination, the procedure returns to the most recent branch point and continues from the second daughter section until there is a termination. This process is illustrated in Figure 5.1.

5.7 Results

The efficacy of the construction procedure is tested by generating 200 samples, in which each sample contains the number of interneurons in the original sample. The results of these simulations are used to construct confidence intervals for parameter and probability density estimates. The original samples contained thirty-one interneurons receiving myelinated input and twenty interneurons receiving unmyelinated input. The simulated samples are used to construct statistics of the properties of the typical interneurons, and therefore can be used to test the efficacy of the construction procedure. Toward this end, two different properties of the sampled dendrites are considered. The first group of properties involve features that are included in the construction process and are referred to as construction-dependent properties. In most cases are univariate densities of diameter or bivariate densities of the relationship between diameters of contiguous sections or of daughter diameters. The second group of properties, referred to as construction-independent properties, are concerned with global features of a simulated dendrite and are not properties that are intrinsic to the process used to generate the sample dendrite. The construction-dependent and -independent properties are listed in Table 5.3.

Properties of the sample of real dendrites	
Distributions used in construction process	Distributions independent of construction process
Number of dendrites	Number of sections
Section lengths	Number of branches
Branching probabilities	Number of branch points
Terminating diameters	Branch length
Contiguous diameters	Dendritic length
Branching diameters	Unbranched dendrites
Parent with 1 daughter	Daughter branch ratios
Parent with 2 daughters	

Table 5.3: Characteristic features to be compared: those used in the construction procedure and those independent of the construction procedure.

One would expect the statistical characteristics of the construction-dependent properties of simulated samples to be identical to that of the original sample. A strong test of the basic

assumption that the section is the basic building block of these interneurons, and that its properties are location independent, would be to show that the construction-independent properties for simulated samples are identical to those of the original sample. Each group of characteristic features will be considered in turn.

5.7.1 First test of the construction procedure

As a simple test of the construction procedure the number of sections, branches, dendrites and branch points from the original sample are compared in Tables 5.4 and 5.5 with those found in the simulations.

Properties of dendrites	Used in simulation	Observed value	Simulated value, mean ± std. dev.
Number of dendrites	Yes	95	95.2 ± 4.6
Number of sections	No	1340	1525.2 ± 163.3
Number of branches	No	494	551.4 ± 60.6
Number of branch points	No	160	143.4 ± 14.9

Table 5.4: A comparison of the elementary properties of dendritic morphology based on the sample of thirty-one interneurons receiving myelinated input with that based on 200 simulations. Each simulation generates a sample the same size as the original sample.

Properties of dendrites	Used in simulation	Observed value	Simulated value, mean ± std. dev.
Number of dendrites	Yes	62	61.8 ± 3.9
Number of sections	No	1179	1176.6 ± 155.6
Number of branches	No	363	349.5 ± 51.9
Number of branch points	No	120	96.2 ± 12.5

Table 5.5: A comparison of the elementary properties of dendritic morphology based on the sample of twenty interneurons receiving unmyelinated input with that based on 200 simulations. Each simulation generates a sample the same size as the original sample.

It is clear from Tables 5.4 and 5.5 that for each comparison the observed value of the dendritic property lies within two standard deviations of its simulated value, thus demonstrating that the properties of the simulated cells are statistically indistinguishable from the real cells. It now remains to compare the original probability densities with the simulated probability densities.

5.7.2 Comparison of observed and simulated probability densities

The observed probability densities are now compared with the probability densities of the construction-dependent and -independent properties. These comparisons are treated separately.

Comparison of observed and construction-dependent probability densities

A crucial feature of the construction process is the simulation of the probabilities of terminating sections, continuing sections and branching sections. These probabilities are calculated from the distributions of the diameters of terminating sections, continuing sections and branching sections. These densities are now compared in Figure 5.7.

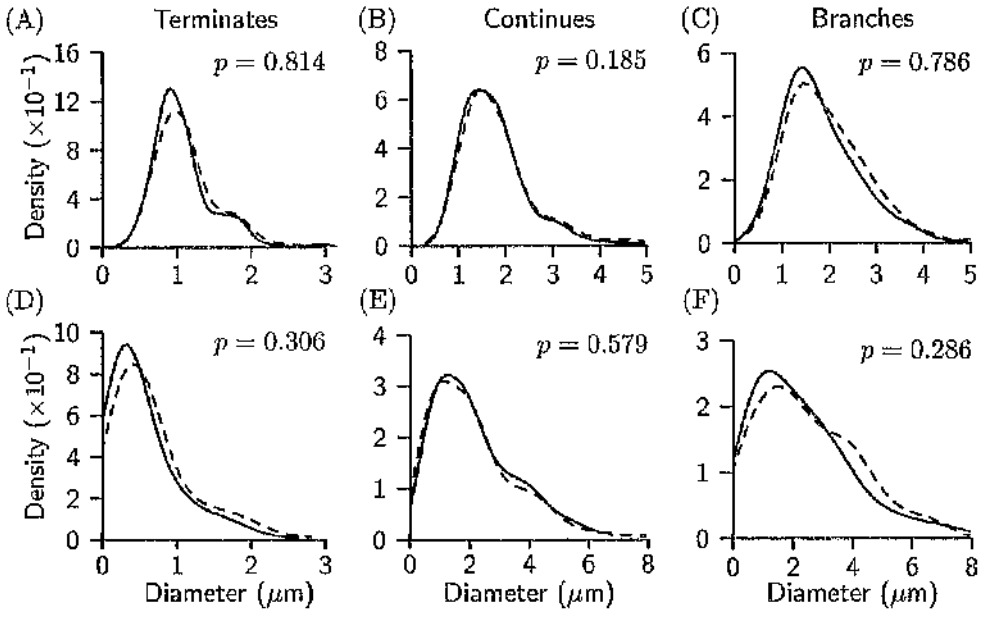

Figure 5.7: The distribution of the diameters of terminating (A) and (D), continuing (B) and (E) and branching (C) and (F) sections for the original (solid line) and simulated (dashed line) samples of interneurons. Graphs (A)-(C) correspond to interneurons receiving myelinated input and graphs (D)-(F) correspond to interneurons receiving unmyelinated input. The p-value for the null hypothesis that the distributions are identical is shown in the upper right-hand corner of each graph.

Figure 5.7 illustrates each of the observed (solid line) and simulated densities (dashed

line) for each type of section ending along with the corresponding p-value for the null hypothesis that the distributions are identical. The figure illustrates the comparisons for interneurons receiving myelinated afferent input (5.7A-C) and unmyelinated afferent input (5.7D-F), and in each case the null hypothesis is not rejected.

There are two further comparisons to be made between the estimated probability densities based on the original sample and the estimated probability densities from the simulation involving construction-dependent properties of the dendrite.

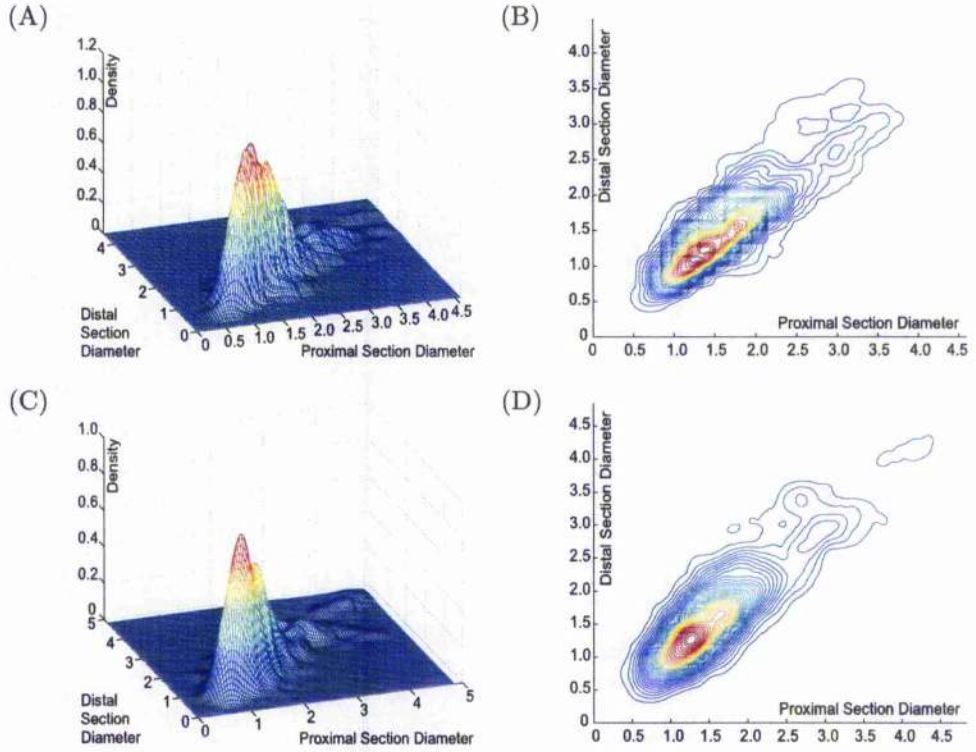

Figure 5.8: Estimated probability densities of contiguous section diameters for interneurons receiving myelinated input. Panels (A) and (C) show surface plots for the original sample and simulated sample respectively, while panels (B) and (D) show the associated contour plots. Components of contiguous sections are referred to as proximal and distal sections. All diameters are measured in μm.

Figure 5.8A,B shows the estimated bivariate probability densities of the diameters of contiguous section of interneurons receiving myelinated afferent input, whereas Figure 5.8C,D illustrates the same densities for simulated samples. It is clear that the simulated

results captures well the qualitative properties of the original sample.

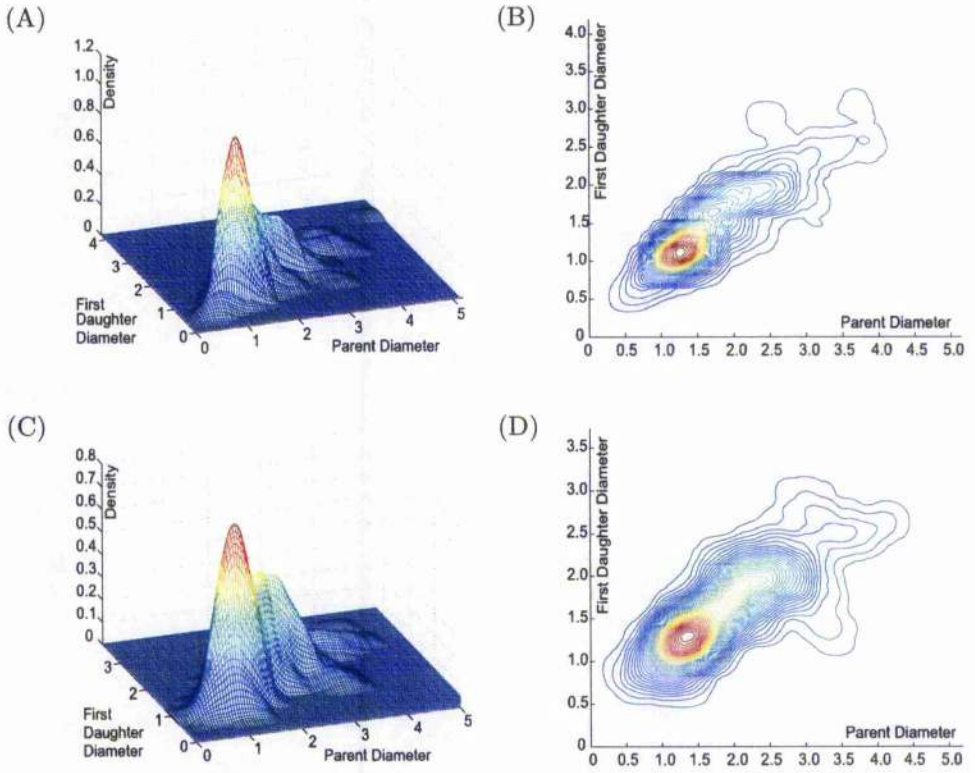

Figure 5.9: Estimated probability densities of parent section diameters and first daughter section diameters for interneurons receiving myelinated input. Panels (A) and (C) show surface plots for the original sample and simulated sample respectively, while panels (B) and (D) show the associated contour plots. All diameters are measured in μm.

Further, Figure 5.9A,B shows the estimated bivariate probability densities of the diameter of parent sections and the diameter of the first daughter section at a branch point for interneurons receiving myelinated afferent input, whereas Figure 5.9C,D shows the same densities for simulated samples. The simulated results give good qualitative agreement with the properties of the original sample.

The results of this section demonstrate that the probability densities used in the construction process are accurately reconstructed by the simulation procedure.

Comparison of observed and construction-independent densities

A strong test of the basic assumption that the section is the basic building block of these interneurons, and that its properties are location independent, is to predict the global features of dendritic morphology that are not intrinsic to the construction process. Recall that it has been demonstrated (Section 5.7.1) that the number of sections, number of branches and number of branch points in the original sample have been shown to be statistically indistinguishable from those in the simulated samples. Other construction-independent features such as daughter-branch ratio, joint distribution of the diameters of daughter branches, characteristic interneuron lengths and unbranched dendrites are considered in this section.

Daughter branch ratio The first comparison of the global properties in this section is between the daughter branch ratio of the original and simulated samples, defined as "the ratio of the diameters of daughter-branch processes" at a branch point (Hillman, 1979). Figure 5.10 illustrates histograms of these distributions for the original and simulated samples of interneurons for both types of afferent input.

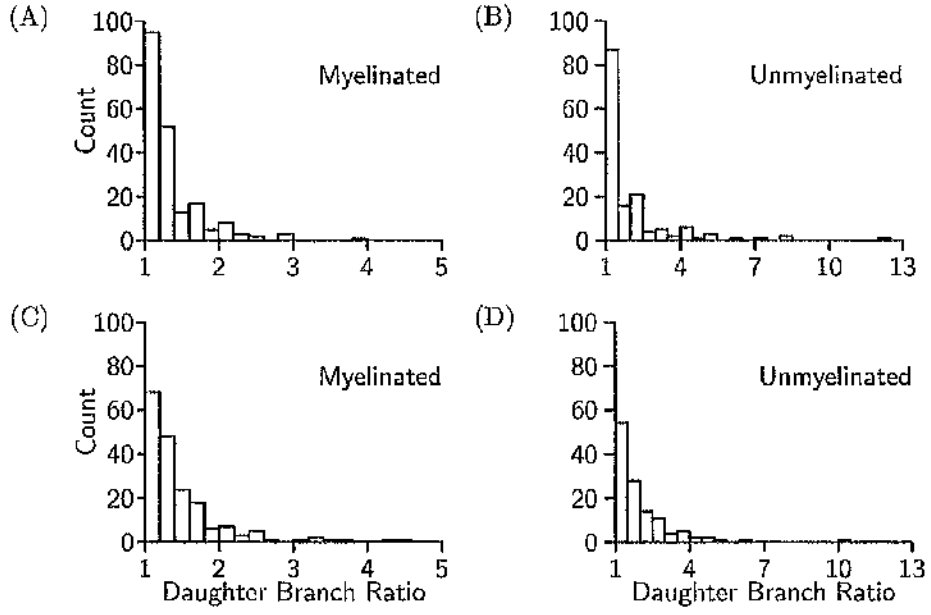

Figure 5.10: Histograms showing the distributions of daughter branch ratios. Panels (A) and (C) show histograms for the original sample and simulated sample respectively for interneurons receiving myelinated input, while panels (B) and (D) show these histograms for interneurons receiving unmyelinated input. Binwidth: (A),(C) = 0.2, (B),(D) = 0.5.

As in the case of the distributions illustrated in the Section 5.7.2, there is good agreement between the histograms generated from the original sample and those generated from the simulation. Moreover, the distributions in Figure 5.10 for the original and simulated samples were found to be statistically indistinguishable. For interneurons receiving myelinated afferent input, the p-value for the null hypothesis that the distributions are indistinguishable was p= 0.901, whereas the p-value for the same null hypothesis for interneurons receiving unmyelinated afferent input was p= 0.806.

Joint distribution of the diameters of daughter sections Figure 5.11A,B shows the estimated bivariate probability densities of the daughter section diameters at a branch point for interneurons receiving myelinated afferent input, whereas Figure 5.11C,D illustrates the same densities for simulated samples. The simulated sample captures well the qualitative properties of the real data.

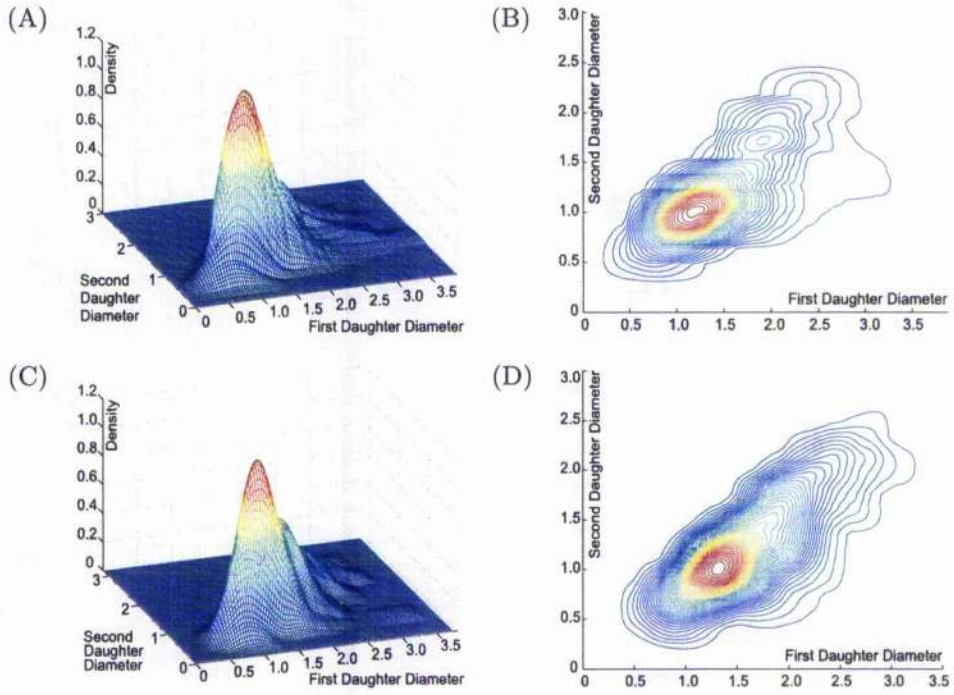

Figure 5.11: Estimated probability densities of the diameter of the daughter sections for interneurons receiving myelinated input. Panels (A) and (C) show surface plots for the original sample and simulated sample respectively, while Panels (B) and (D) show the associated contour plots. All diameters are measured in μm.

Characteristic neuronal lengths The final comparison of the global properties between the original and simulated samples will consider the distributions of section, branch and dendritic length for interneurons receiving myelinated afferent input. Recall that there were 31 interneurons receiving myelinated afferent input. Thus 200 simulations each generating 31 interneurons was carried out. These simulations allow an estimate of the mean value and standard deviation of the value of the density of each characteristic length for the simulated interneurons. Figure 5.12 shows the mean value (dashed line) and two standard deviations about the mean value (dotted lines) for these simulated probability densities. The solid line in each panel in Figure 5.12 shows the estimated probability density for the original sample.

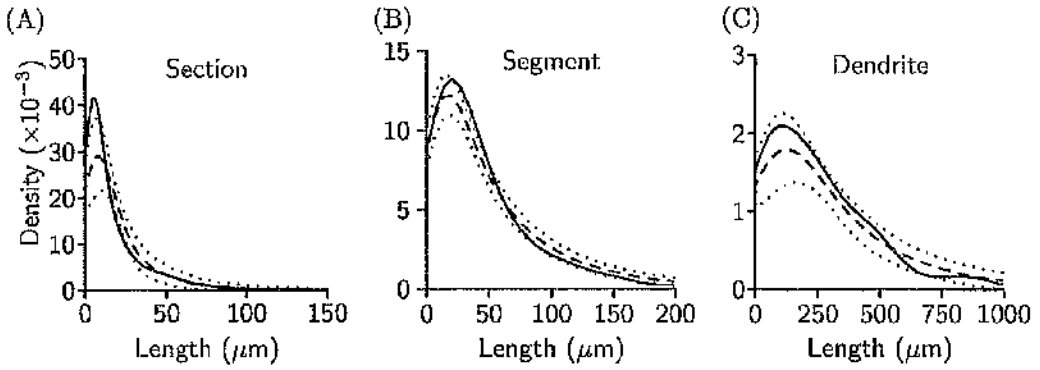

Figure 5.12: Each graph displays the distribution of lengths from the original interneurons (solid line), the mean distribution of simulated lengths (dashed line) $\pm$ two standard deviations (dotted line). Graphs (A), (B) and (C) correspond to the section, segment and dendrite length respectively, from interneurons receiving myelinated input.

With the exception of the peak value for the density of section lengths (Figure 5.12A), all values of the density based on the original sample lie within two standard deviations of the mean of the simulated density. These figures clearly demonstrate that the construction procedure successfully predicts the characteristic lengths of the dendrite. Interneurons that receive unmyelinated afferent input are treated subsequently.

Unbranched dendrites Approximately a quarter/sixth of all dendrites from the sample of interneurons with myelinated/unmyelinated input were unbranched, that is, they grew from the soma and terminated without branching. Recall that these dendrites were excluded from the analysis in Section 5.5 to avoid issues of bias. However the sections

from these dendrites were included in the simulation process. Unbranched dendrites are generated naturally by the simulation process and their proportion in a simulated sample serves as an independent test of the basic concept that the underlying properties of a dendritic section is independent of its location within a dendritic tree.

Afferent input	No. of Dendrites	No. of Unbranched	Percentage
Myelinated	95	23	24.2%
Sim Myelinated	95.2 ± 4.6	30.8 ± 4.8	32.4% ± 4.6
Unmyelinated	62	11	17.7%
Sim Unmyelinated	61.8 ± 3.9	18.6 ± 3.6	30.3% ± 6.3

Table 5.6: Ratios of unbranched to branched dendrites for the original and simulated samples of interneurons receiving myelinated and unmyelinated input. Simulated values are described by the mean ± standard deviation.

The proportion of unbranched dendrites in the original sample lies within two standard deviations of the mean value predicted by simulation.

Dendograms

It has been demonstrated that the properties of the simulated interneurons receiving myelinated afferent input are statistically indistinguishable from those of the original sample. Figure 5.13 illustrates the dendogram for one of the sample interneurons and a simulated interneuron.

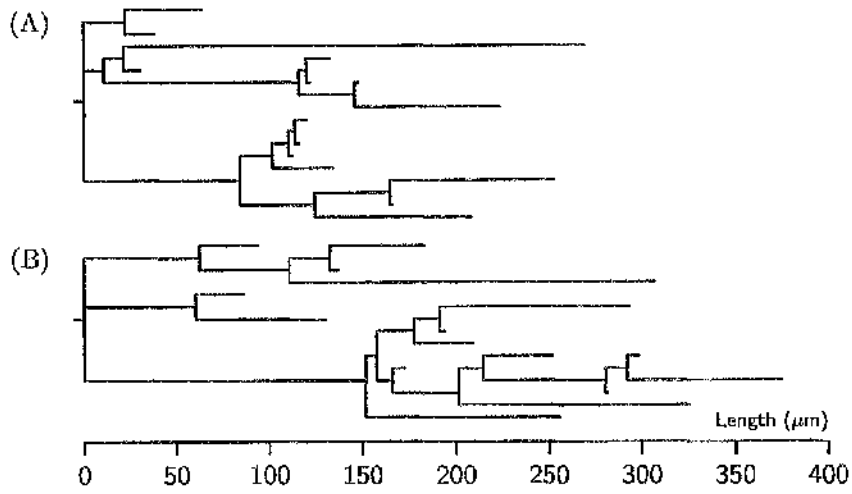

Figure 5.13: Two example dendograms of interneurons that receive myelinated input, where dendograms (A) and (B) are from the original and simulated samples respectively.

It is clear from Figure 5.13 that the original and simulated interneurons receiving myelinated input display similar branching patterns: some short clustered branches and other long unbranched segments. Furthermore, the unusual feature of very short branches after a branch point is common in both the original and simulated interneurons. Quantitatively, the original interneuron receiving myelinated input (Figure 5.13A) has 27 branches and 12 branch points, while the simulated interneuron (Figure 5.13B) has 29 branches and 13 branch points.

5.7.3 Summary

The underlying assumption of the construction procedure, namely that the joint distribution of the diameter and lengths of dendritic sections is independent of position in the dendritic tree is validated. Both the construction-dependent and -independent properties of the interneurons are retrieved by the recursive algorithm used to simulate a dendritic tree, further validating the basic assumption.

5.7.4 A further development

A feature of the dendrites has been identified that requires special treatment in the simulation procedure. Specifically, the calculation of the bandwidth for the kernel density estimation procedures assumes that the underlying density is unimodal.

The assumption of a unimodal density

Figure 5.14 shows the mean value (dashed line) and two standard deviations about the mean value (dotted lines) for the estimated probability densities of simulated section, segment and dendritic lengths from interneurons receiving unmyelinated input. The solid line in each panel of Figure 5.14 shows the estimated probability density for the original sample.

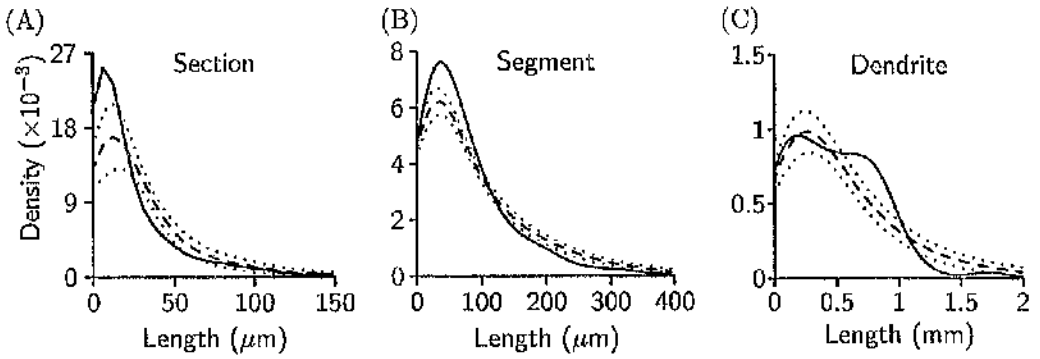

Figure 5.14: Each graph displays the distribution of lengths from the original interneurons (solid line), the mean distribution of simulated lengths (dashed line) ± two standard deviations (dotted line). Graphs (A), (B) and (C) correspond to the section, segment and dendrite length respectively, from interneurons receiving unmyelinated input.

The fit between the original and simulated samples for interneurons receiving unmyelinated input is less successful than that for the interneurons receiving myelinated input (see Figure 5.12). In fact, Figure 5.14C suggests that the underlying density of dendritic lengths may not be unimodal, which in turn suggests that interneurons with unmyelinated input have two types of dendrites, namely long dendrites and short dendrites. Although Olave *et al.* (2002) suggested this might be the case, they were unable to quantify the difference. If this is truly the case then both classes of dendrite must be identified and then treated separately. This is an issue for further work.

5.8 Conclusions

The procedure described in this chapter for simulating dendrites differs in two important ways from previous methods. First, the basis of the simulation is the single assumption that the joint distribution of diameter and length of a dendritic section is independent of its location within a dendritic tree. That is, the distribution of length for dendritic sections with the same diameter drawn from different parts of the tree are identical. Second, the procedure does not assume specific parametric forms for the probability densities characterising the features of a dendrite, but instead uses the kernel density estimation technique to provide a non-parametric estimate of these densities.

The simplicity of the approach has resulted in a straightforward simulation procedure that successfully generates dendritic trees that are statistically indistinguishable from the original sample.

Chapter 6

Concluding remarks and future work

The work of this thesis contributes to three different areas of mathematical modelling in neuroscience. Each contribution has introduced a novel approach to a particular neurophysiological problem. The first problem focused on the discrepancy between the observed and predicted conduction speed of the propagated action potential, and the extent to which the effects of biological variability and measurement error can account for this discrepancy. The second problem considered the analytical development of the equivalent cable. In this analysis an arbitrary branched structure was represented by a piecewise uniform cable and a unique bijective mapping of input between the branched structure and the cable. The third problem developed a new and parsimonious procedure for the simulation of dendritic morphology based on the estimation of probability density by the kernel method.

With respect to the Hodgkin-Huxley membrane model, measurement error and biological variability were eliminated as possible sources of the discrepancy in conduction speed in Chapter 2. Work by Armstrong and Bezanilla (1977), Bezanilla and Armstrong (1977) and more recently by Clay (1998) suggests that the discrepancy might be accounted for by a revision of the models for the sodium and potassium kinetics. Although Clay considers how the threshold for the action potential and its latency are affected by the revised kinetic scheme, he does not use the new kinetics to predict the conduction speed of the propagated action potential and compare it with its observed value. This would be the most important test of the new model since it is a test which is independent of the procedure used to derive

the kinetic scheme. In a future study, the revised kinetics scheme will be used to predict the conduction speed of the propagated action potential.

A complete description of the procedure used to derive the equivalent cable was given in Chapter 4. This procedure transforms an arbitrary branched structure into an equivalent unbranched cable. The main feature of this procedure is the bijective mapping that exists between the branched and unbranched structures. In a future study, the bijective mapping will be used as a tool to investigate the influence of the neuronal morphology on neuronal behaviour.

A new procedure for simulating neuronal morphology was described in Chapter 5. This procedure was based on a single assumption that the joint distribution of diameter and length of a dendritic section is independent of its location within a dendritic tree. By using univariate and multivariate kernel density estimation procedures, samples of spinal interneurons were simulated successfully from a sample of interneurons with myelinated afferent input. Future work will involve the refinement of this procedure to estimate accurately bimodal distributions and also its application to other classes of neurons.

References

Armstrong, C.M. and Bezanilla, F. (1977). Inactivation of the sodium channel: II. Gating current experiments. *J. Gen. Physiol.* **70**, 567-590.

Ascoli, G. A. and Krichmar, J. L. (2000). L-neuron: A modelling tool for the efficient generation and parsimonious description of dendritic morphology. *Neurocomputing.* **32-33**, 1003-1011.

Ascoli, G. A., Krichmar, J. L., Nasuto, S. J. and Senft, S. L. (2001). Generation, description and storage of dendritic morphology data. *Phil. Trans. R. Soc. Lond.* B. **356**, 1131-1145.

Barrett, J.N. and Crill, W.E. (1974). Specific membrane properties of cat motoneurones. *J. Physiol.* **239**, 301-324.

Bezanilla, F., and Armstrong, C.M. (1977). Inactivation of the sodium channel: I. Sodium current experiments. *J. Gen. Physiol.* **70**, 549-566.

Burke, R. E., Marks, W. B. and Ulfhake, B. (1992). A parsimonious description of motoneuron dendritic morphology using computer simulation. *J. Neurosci.* **12**, 2403-2416.

Ramón y Cajal, S. (1952). Histologie du Systèm Nerveux de l'Homme & des Vertébrés, CSIC, Madrid.

Caterall, W.A. (1988). Structure and function of voltage-sensitive ion channels. *Science.* **242**, 50-61.

Caterall, W.A. (1992). Cellular and molecular biology of voltage-gated sodium channels. *Physio. Rev.* **72**(Supp), S15-S48.

Clay, J. R. (1998). Excitability of the squid giant axon revisited. *J. Neurophysiol.* **80**, 903-913.

Clements, J.D. and Redman, S.J. (1989). Cable properties of cat spinal motoneurons measured by combining voltage clamp, current clamp and intra-cellular staining. *J. Physiol.* **409**, 63-87.

Cole, K.S. (1968). *Membranes, Ions and Impulses.* University of California Press: Berkeley and Los Angeles.

Cooley, J. W. and Dodge Jr., F. A. (1966). Digital computer solutions for excitation and propagation of the nerve impulse. *Biophys. J.* **6**, 583-599.

Cullheim, S., Fleshman, J. W., Glenn, L. L. and Burke, R. E. (1987). Three-dimensional architecture of dendritic trees in type-identified α-motoneurons. *J. Comp. Neurol.* **255**, 82-96.

Davis, L. Jr and Lorente de Nó, R. (1947). Contribution to the mathematical theory of the electrotonus. *Stud. Rockefeller Inst. M. Res.* **131**, 442-496.

Devaud, J. M., Quenet, B., Gascuel, J. and Masson, C. (2000). Statistical analysis and parsimonious modelling of dendograms of *in vitro* neurons. *Bull. Math. Biol.* **62**, 657-683.

Golub, G.H. and Van Loan, C.F. (1989). *Matrix Computations* (2nd Ed.). Johns Hopkins University Press, Baltimore.

Hermann, L. (1884). Nachtrag zu Seite 150. *Pflügers Arch. ges. Physiol.* **33**, 162-168.

Hillman, D. E. (1979). Neuronal shape parameters and substructures as a basis of neuronal form. In: *The neurosciences, fourth study program.* Ed. Schmitt, F. pp. 477-498. Cambridge, MA: MIT Press.

Hodgkin, A.L. and Huxley, A. F. (1939). Action potentials recorded from inside a nerve fibre. *Nature.* **144**, 710-711.

Hodgkin, A.L. and Rushton, W.A.H. (1946). The electrical constants of a crustacean nerve fibre. *Proc. Royal Soc. Lond.* **B 133**, 444-479.

Hodgkin, A.L. and Huxley, A. F. (1952a). Currents carried by sodium and potassium through the membrane of the giant axon of *Loligo. J. Physiol.* **116**, 449-472.

Hodgkin, A.L. and Huxley, A. F. (1952b). The components of membrane conductance in the giant axon of *Loligo. J. Physiol.* **116**, 473-496.

Hodgkin, A.L. and Huxley, A. F. (1952c). The dual effect of membrane potential on sodium conductance in the giant axon of *Loligo*. *J. Physiol.* **116**, 497-506.

Hodgkin, A. L. & Huxley, A. F. (1952d). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.* **117**, 500-544.

Hodgkin, A. L., Huxley, A. F. & Katz, B. (1952). Measurements of current-voltage relations in the membrane of the giant axon of *Loligo*. *J. Physio.* **116**, 424-448.

Hoel, P.G. (1954). *Introduction to Mathematical Statistics*. Wylie, NY.

Holmes, W.R. (1986). A continuous cable method for determining the transient potential in passive dendritic trees of known geometry. *Biol. Cybern.* **55**, 115-124.

Huxley, A. F. (1959). Ion movements during nerve activity. *Ann. NY Acad. Sci.* **81**, 221-246.

Jack, J.J.B., Noble, D. and Tsien, R.W. (1975). *Electric current flow in excitable cells.* OUP, Oxford.

Jack, J. and Redman, S. (1995). Introduction to cable properties of neurons with complex dendritic trees. In: *The Theoretical Foundation of Dendritic Function — Selected Papers of Wilfrid Rall with Commentaries.* Ed. Segev, I , Rinzel, J. and Shepherd, G.M. pp. 27-33. The MIT Press, Cambridge, Massachusetts.

Jankowska, E. (1992). Interneuronal relay in spinal pathways from propioceptors. *Prog. Neurobiol.* **38**, 335-378.

Kaspirzhny, A.V., Gogan, P. Horcholle-Bossavit, G. and Tyč-Dumont, S. (2002). Neuronal morphology data bases: morphological noise and assesment of data quality. *Network: Comput. Neural Syst.* **13**, 357-380.

Katz, B. (1966). *Nerve, muscle and synapse.* McGraw-Hill. New York.

Kelvin, Lord (William Thompson). (1855). On the theory of the electric telegraph. *Proc. R. Soc. Lond.* **7**, 382-399.

Koch, C., Poggio, T. and Torre, V. (1982). Retinal ganglion cells: A functional interpretation of dendritic morphology. *Phil. Trans. R. Soc. Lond.* **B 298**, 227-263.

Kreusch, A., Pfalfinger, P. J., Stevens, C. F. & Choe, S. (1998). Crystal structure of the tetramerization domain of the *Shaker* potassium channel. *Nature.* **392**, 945-948.

Larkman, A. U. (1991). Dendritic morphology of pyramidal neurons of the visual cortex of the rat: I. Branching Patterns. *J. Comp. Neurol.* **306**, 307-319.

Lindenmayer, A. (1968). Mathematical models for cellular interactions in development I & II. *J. Theor. Biol.* **18**, 280-315.

Lindsay, K.A., Ogden, J.M., Halliday, D.M. and Rosenberg, J.R. (1999). An introduction to the principles of neuronal modelling. In: *Modern techniques in neuroscience research.* Ed. Windhorst, U. and Johannson, H. pp. 213-306. Springer Verlag, Berlin.

Lindsay, K.A., Ogden, J.M. and J.R. Rosenberg, J.R. (2001)a. Dendritic subunits determined by dendritic morphology. *Neural Comp.* **13**, 2465-2476.

Lindsay, K.A., Ogden, J.M. and Rosenberg, J.R. (2001)b. Equivalence transformations for dendritic Y-junctions: a new definition of dendritic sub-unit. *Math. Biosci.* **170**, 133-154.

Mainen, Z.F. and Sejnowski, T.J. (1996). Influence of dendritic structure on firing pattern in model neocortical neurons. *Nature* **382**, 363-366.

Mel, B.W. (1994). Information processing in dendritic trees. *Neural Comp.* **6**, 1031-1085.

Miller, R.N. and Rinzel, J. (1981). The dependence of impulse propagation speed on firing frequency, dispersion, for the Hodgkin-Huxley model. *Biophys. J.* **34**, 227-259.

Meunier, C. and Segev, I. (2002). Playing the devil's advocate: is the Hodgkin-Huxley model useful? *Trends Neurosci.* **25**, 558-563.

Ogden, J.M., Rosenberg, J.R. and Whitehead, R.R. (1999). The Lanczos procedure for generating equivalent cables. In: *Modelling in the Neurosciences - from ionic channels to neural networks.* Ed. Poznanski, R.R. pp. 177-229. Harwood Academic Publishers, Australia.

Olave, M. J., Puri, N., Kerr, R. and Maxwell, D. J. (2002). Myelinated and unmyelinated primary afferent axons form contacts with cholinergic interneurons in the spinal dorsal horn. *Exp. Brain Res.* **145**, 448-456.

Pallotta, B.S. and Waggoner, P.K. (1992). Voltage-dependent potassium channels since Hodgkin and Huxley. *Physio. Rev.* **72**(Supp), S49-S67.

Rall, W. (1959). Branching dendritic trees and motoneuron membrane resistivity. *Exp. Neurol.* **1**, 491-527.

Rall, W. (1962). Theory of physiological properties of dendrites. *Ann. N. Y. Acad. Sci.* **96**, 1071-1092.

Sato, C., Ueno, Y., Asai, K., Takahashi, K., Sato, M., Engel, A. & Fujiyoshi, Y. (2001). The voltage-sensitive sodium channel is a bell-shaped molecule with several cavities. *Nature.* **409**, 1047-1051.

Scott, D.W. (1992). *Multivariate Density Estimation Theory, Practice, and Visualisation.* John Wiley and Sons, Inc, New York.

Segev, I. and Burke, R.E. (1989). Compartmental models of complex neurons. In: *Methods in neuronal modelling: from synapses to networks.*Ed. Koch, C. and Segev, I. pp. 93-136. MIT press, Cambridge, MA.

Segev, I. (1992). Single neurone models: oversimple, complex and reduced. *Trends Neurosci.* **15**, 414-21.

Shepherd, G.M. and Brayton, R.K. (1987). Logic operations are properties of computer-simulated interactions between excitable dendritic spines. *Neuroscience.* **21**, 151-166.

Silverman, B. W. (1986). *Density estimation for statistics and data analysis.* London: Chapman & Hall.

Tamori, Y. (1993). Theory of dendritic morphology. *Phys. Rev.* E **48**, 3124-3129.

Uemura, E., Carriquiry, A., Kliemann, W. and Goodwin, J. (1995). Mathematical modelling of dendritic growth in vitro. *Brain Res.* **671**, 187-194.

Ulfhake, B. and Kellerth, J.-O. (1983). A quantitative morphological study of HRP-labelled cat α-motoneurones supplying different hindlimb muscles. *Brain Res.* **264**, 1-19.

Ulfhake, B. and Kellerth, J.-O. (1984). Electrophysiological and morphological measurements in cat gastrocnemius and soleus α-motoneurons. *Brain Res.* **307**, 167-179.

Van Pelt, J. (1992). A simple vector implementation of the Laplace-transformed cable equations in passive dendritic trees. *Biol. Cybern.* **68**, 15-21.

Van Pelt, J. and Uylings, H. B. M. (1999). Natural variability in the geometry of dendritic branching patterns. In: *Modelling in the neurosciences: from ionic channels to neural networks.* Ed. Poznanski, R. R. pp. 79-108. Amsterdam, The Netherlands: Harwood Academic Publishing.

Van Pelt, J. and Uylings, H. B. M. (2002). Branching rates and growth functions in the outgrowth of dendritic branching patterns. *Network: Comp. Neur. Syst.* **13**, 261-281.

Vetter, P., Roth, A. and Häusser, M. (2001). Propagation of action potentials in dendrites depends on dendritic morphology. *J. Neurophys.* **85**, 926-937.

Whitehead, R.R. and Rosenberg, J.R. (1993). On trees as equivalent cables. *Proc. R. Soc. Lond.* **B 252**, 103-108.

Appendix A

Associated programs

Each of the projects described in the previous chapters use C programming to implement the various mathematical problems and manipulations. Therefore the development of the code is itself a significant component of each project and is included here to compliment each chapter. To avoid repetition, those functions used regularly are brought together at the end of this appendix after the main body of programs.

A.1 Action potentials

Despite using the same model, namely the Hodgkin-Huxley membrane model, the unique problems of estimating the conduction velocity of a propagating action potentials and the dispersion characteristics of a train of action potentials had to be treated quite differently. Therefore, two distinct programs had to be developed to solve each problem.

A.1.1 HHPotVel.c

The first, HHPotVel.c investigates the conduction speed of the action potential after a current has been injected into an axon at the resting membrane potential. In this case, if the injected current is large enough, the action potential develops and then propagates away from the point of stimulation.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
```

```
/*****************************************************************
          Travelling Waves - Hodgkin Huxley - Squid Giant Axon

              SOLVES THE PERIODIC HODGKIN HUXLEY EQUATIONS
                  USING SPECTRAL METHOD - FOURIER

          C_m V_t + J = (g_A*R/2) V_xx  V(x,0) = F(x) given

  AND   (a) Spectral method - majority of program carried out in
            coefficent space - converting back for solution.
        (b) Resting axon is given a large injected current, and
            initiates two propagating waves: investigating first
            moments after initialisation.

  This program calculates & prints out:
       (i) core current along axon                    core.dat
      (ii) radial current out of axon                 radial.dat
     (iii) velocity of left & right hand waves        velocity.dat
      (iv) potential & channel kinetic values         hhwave##
       (v) z-values: axial coordinates                zval.dat
      (vi) r-values: radial coordinates               rval.dat
     (vii) intracellular potentials across radii      intpot.dat
    (viii) extracellular potentials to "infinity"     extpot.dat
*****************************************************************/

/*  Declaration of Global functions */
double bess_i0(double);
double bess_i1(double);
double bess_k0(double);
double bess_k1(double);
void fprime( double, double *, double *);
void sgsolve( int, int *, double *, double *, double *, double, double *,
                      void (*fcn)( double, double *, double *), int *);
void intrp( int, double, double *, double, double *, double *, int, double **, double *);
void step( int, double *, double *, double *, double *, double *, double *, int *, int *,
           int *, int *, double **, double *, double *, double *, double, double,
           void (*fcn)(double, double *, double *));
void real_c( int, double *, double *);
void real_v( int, double *, double *);

/*  Biological Parameters */
#define    CELSIUS      18.5        /* Celsius temperature of neuron */
#define         RD     0.0238        /* Radius of squid axon (cm) */
#define         GA     28.249        /* Specific intra-cell'r conductance (mmho/cm)*/
#define         GE     28.249        /*Specific extra-cell'r conductance (mmho/cm)*/
#define         CM       1.0         /*Specific membrane capacitance (muF/cm^2)*/

/*  Parameters for ODE solver */
#define          N       1600        /* Must be even */
#define       TEND        0.6        /* Final time */
#define         DT       0.01        /* Time step */
#define        TOL     5.e-16        /* Error tolerance */
#define        LEN       16.0        /* Window length */

/*  Other parameters */
#define       FRAC        1.0        /* Fraction of v_na for injected current*/
#define        DIV        100        /* Divisions of RD */
#define        FAC       1.05        /* Multiplication factor for extracellular region */
#define        NUM          9        /* Number of potentials to be sampled */

/*  Declaration of HH coefficient functions */
```

```
double alfa_h( double );
double alfa_m( double );
double alfa_n( double );
double beta_h( double );
double beta_m( double );
double beta_n( double );

/*  Derivative of Hodgkin Huxley functions */
double d_alfa_h(double);
double d_alfa_m(double);
double d_alfa_n(double);
double d_beta_h(double);
double d_beta_m(double);
double d_beta_n(double);

double *fac, *rad;

void main(void)
{
    extern double *fac, *rad;
    double dx=0.01, *x, *y, *c, *cc, *cur, *nu, *icof, *ecof, *e, tnew, told, *ep, dist,
           ti, to, tol, relerr, abserr, rvl, lvl, ivdc, lvm, lvr, rvm, rvr, ir, il, tf,
           ts, xlf, xrf, xls, xrs, frac, xold[NUM], xnew[NUM], *mp, tmp, pi, veq, meq,
           heq, neq, vn, vu, vl, vr, pos;
    double pots[NUM]={-50.0, -25.0, 0.0, 10.0, 20.0, 25.0, 28.0, 29.0, 30.0};
    char filename[20], ext[3];
    int j, k, nh, ndim, ifail, fig, im, kk, setmem=1, first, append, append1, append2,
        success, start, first1;
    static double v_na=55.0, v_k=-72.0, v_l=-49.387, g_na=120.0, g_k=36.0, g_l=0.3;
    FILE *fp, *fp1;

/*********************************************************
    Variables
    ---------
            y[0] -> y[N-1]      Holds the potential
            y[N] -> y[2*N-1]    Holds H at nodes
          y[2*N] -> y[3*N-1]    Holds M at nodes
          y[3*N] -> y[4*N-1]    Holds N at nodes
*********************************************************/

/*  STEP 1. - Build ratio of Bessel functions */
    nh = N/2;
    pi = 4.0*atan(1.0);

    nu = (double *) malloc( (nh+1)*sizeof(double) );
    icof = (double *) malloc( (nh+1)*sizeof(double) );
    ecof = (double *) malloc( (nh+1)*sizeof(double) );
    fac = (double *) malloc( (nh+1)*sizeof(double) );
    cur = (double *) malloc( (nh+1)*sizeof(double) );
    rad = (double *) malloc( (DIV+1)*sizeof(double) );
    fac[0] = 0.0;
    cur[0] = 0.0;
    nu[0] = 0.0;
    icof[0] = 1.0;
    ecof[0] = 0.0;
    for ( k=0 ; k<=DIV ; k++ ) rad[k] = (RD/DIV)*((double) k);
    for ( k=1 ; k<=nh ; k++ ) {
        tmp = 2.0*pi*RD*((double) k)/LEN;
        nu[k] = tmp;
        icof[k] = GE*bess_k1(tmp)/(GE*bess_k1(tmp)*bess_i0(tmp)+GA
                                                    *bess_k0(tmp)*bess_i1(tmp));
```

```
        ecof[k] = -GA*bess_i1(tmp)/(GE*bess_k1(tmp)*bess_i0(tmp)+GA
                                                    *bess_k0(tmp)*bess_i1(tmp));
        fac[k] = GE*tmp*bess_k1(tmp)*bess_i1(tmp)/(GE*bess_k1(tmp)*bess_i0(tmp)+GA
                                                    *bess_k0(tmp)*bess_i1(tmp));
        cur[k] = -GA*RD*GE*bess_k1(tmp)*bess_i1(tmp)/(GE*bess_k1(tmp)*bess_i0(tmp)+GA
                                                    *bess_k0(tmp)*bess_i1(tmp));
    }

/*  STEP 2. - Allocate memory to hold file information */
    first = 1;
    first1 = 1;
    start = 1;
    ndim = 4*N;
    x = (double *) malloc( (N+1)*sizeof(double) );
    y = (double *) malloc( ndim*sizeof(double) );
    mp = (double *) malloc( N*sizeof(double) );
    c = (double *) malloc( N*sizeof(double) );
    cc = (double *) malloc( N*sizeof(double) );
    e = (double *) malloc( N*sizeof(double) );
    ep = (double *) malloc( N*sizeof(double) );

/*  STEP 3. - Calculate equilibrium potential */
    vn = -62.0;
    vu = -58.0;
    do {
        veq = 0.5*(vn+vu);
        heq = alfa_h(veq)/(alfa_h(veq)+beta_h(veq));
        meq = alfa_m(veq)/(alfa_m(veq)+beta_m(veq));
        neq = alfa_n(veq)/(alfa_n(veq)+beta_n(veq));
        ivdc = g_na*pow(meq,3)*heq*(veq-v_na)+g_k*pow(neq,4)*(veq-v_k)+g_l*(veq-v_l);
        if ( ivdc < 0.0 ) {
            vn = veq;
        } else {
            vu = veq;
        }
    } while ( vu-vn > 5.e-7 );

/*  STEP 4. - Initialise the membrane */
    for ( k=0 ; k<N ; k++ ) {
        x[k] = dx*((double) k);
        mp[k] = veq;
        y[N+k] = heq;
        y[2*N+k] = meq;
        y[3*N+k] = neq;
    }
    x[N] = LEN;

/*  STEP 5. - Apply injected current to small region of membrane*/
    mp[N/2-5] = FRAC*v_na;
    mp[N/2-4] = FRAC*v_na;
    mp[N/2-3] = FRAC*v_na;
    mp[N/2-2] = FRAC*v_na;
    mp[N/2-1] = FRAC*v_na;
    mp[N/2] = FRAC*v_na;
    mp[N/2+1] = FRAC*v_na;
    mp[N/2+2] = FRAC*v_na;
    mp[N/2+3] = FRAC*v_na;
    mp[N/2+4] = FRAC*v_na;
    mp[N/2+5] = FRAC*v_na;

/*  STEP 6. - Compute coefficients of initial voltage profile */
```

```
    real_c(N, mp, y);

/*  STEP 7. - Integrate forward in time */
    ti = 0.0;
    fig = 1;
    while ( ti < TEND ) {
        to = ti+DT;
        relerr = 5.e-16;
        abserr = 5.e-16;
        ifail = -1;
        sgsolve( ndim, &setmem, &relerr, &abserr, &ti, to, y, fprime, &ifail);
//        if ( count %5 == 0 ) {

/*  STEP 8. - Build voltage profile */
            real_v(N, y, mp);

/*  STEP 9. - Check for potential above pots[NUM] */
            k = nh;
            do {
                success = ( pots[NUM-1] < mp[k] );
                k++;
            } while ( !success && k != N );

            if ( success ) {
                for ( k=0 ; k<NUM ; k++ ) {
                    j = N-1;
                    while ( mp[j] < pots[k] ) j--;
                    frac = (pots[k]-mp[j+1])/(mp[j]-mp[j+1]);
                    xnew[k] = x[j+1]-(x[j+1]-x[j])*frac;
                    tnew = ti;
                }
                if ( first ) {
                    for ( k=0 ; k<NUM ; k++ ) xold[k] = xnew[k];
                    told = tnew;
                    first = 0;
                } else {                // Calculate & print velocity results to file.
                    if ( start ) {
                        fp = fopen("velocity.dat", "w");
                        fp1 = fopen("dist.dat","w");
                        for ( k=0 ; k<NUM ; k++ ) {
                            fprintf(fp, "%8.5lf\t", (xnew[k]-xold[k])/(tnew-told));
                            fprintf(fp1,"%8.5lf\t", xnew[k]-x[nh]);
                        }
                        fprintf(fp,"\n");
                        fprintf(fp1,"\n");
                        fclose(fp);
                        fclose(fp1);
                        for ( k=0 ; k<NUM ; k++ ) xold[k] = xnew[k];
                        told = tnew;
                        start = 0;
                    } else {
                        fp = fopen("velocity.dat", "a");
                        fp1 = fopen("dist.dat","a");
                        for ( k=0 ; k<NUM ; k++ ) {
                            fprintf(fp, "%8.5lf\t", (xnew[k]-xold[k])/(tnew-told));
                            fprintf(fp1,"%8.5lf\t", xnew[k]-x[nh]);
                        }
                        fprintf(fp,"\n");
                        fprintf(fp1,"\n");
                        fclose(fp);
                        fclose(fp1);
```

```
                    for ( k=0 ; k<NUM ; k++ ) xold[k] = xnew[k];
                    told = tnew;
                }
                if ( first1 ) {
                    fp = fopen("times.dat","w");
                    fprintf(fp,"%5.3lf\n", ti);
                    fclose(fp);
                    first1 = 0;
                } else {
                    fp = fopen("times.dat","a");
                    fprintf(fp,"%5.3lf\n", ti);
                    fclose(fp);
                }
            }
        }

/*  STEP 10. - Construct output file names */
            ext[0] = fig/10+48;
            ext[1] = fig%10+48;
            ext[2] = '\0';
            strcpy(filename,"hhwave");
            strcat(filename,ext);
            fp = fopen(filename,"w");
            for ( k=0 ; k<N ; k++ ) {
                fprintf(fp,"%8.5lf\t %8.5lf\t %8.5lf\t %8.5lf\t %8.5lf\n", x[k], mp[k],
                                                        y[k+N], y[2*N+k], y[3*N+k]);
            fprintf(fp,"%8.5lf\t %8.5lf\t %8.5lf\t %8.5lf\t %8.5lf\n", x[N], mp[0],
                                                            y[N], y[2*N], y[3*N]);
            fclose(fp);
            printf("\n Reached %4.2lf",ti);
            fig++;
        }
    }

/*  STEP 11. - Write out file of x,y,z values */
    append = 0;
    for ( k=0 ; k<=DIV ; k++ ) {

        if ( append ) {                             // z-values
            fp = fopen("zval.dat","a");
            for ( j=0 ; j<=N ; j++ ) fprintf(fp,"%9.4lf",x[j]);
            fprintf(fp,"\n");
            fclose(fp);
        } else {
            fp = fopen("zval.dat","w");
            for ( j=0 ; j<=N ; j++ ) fprintf(fp,"%9.4lf",x[j]);
            fprintf(fp,"\n");
            fclose(fp);
        }

        if ( append ) {                             // r-values
            fp = fopen("rval.dat","a");
            for ( j=0 ; j<=N ; j++ ) fprintf(fp,"%12.6lf",rad[k]);
            fprintf(fp,"\n");
            fclose(fp);
        } else {
            fp = fopen("rval.dat","w");
            for ( j=0 ; j<=N ; j++ ) fprintf(fp,"%12.6lf",rad[k]);
            fprintf(fp,"\n");
            fclose(fp);
        }
```

```
/*  STEP 12. - Calculate intracellular potentials over radii */
        c[0] = y[0]*icof[0]*bess_i0(nu[0]*rad[k]);
        c[1] = y[1]*icof[nh]*bess_i0(nu[nh]*rad[k]);
        for ( j=1 ; j<nh ; j++ ) {
            c[2*j] = y[2*j]*icof[j]*bess_i0(nu[j]*rad[k]);
            c[2*j+1] = y[2*j+1]*icof[j]*bess_i0(nu[j]*rad[k]);
        }
        real_v(N, c, mp);

        if ( append ) {                         // intracellular potentials
            fp = fopen("intpot.dat","a");
            for ( j=0 ; j<N ; j++ ) fprintf(fp,"%12.6lf", mp[j]);
            fprintf(fp,"%12.6lf", mp[0]);
            fprintf(fp,"\n");
            fclose(fp);
        } else {
            fp = fopen("intpot.dat","w");
            for ( j=0 ; j<N ; j++ ) fprintf(fp,"%12.6lf", mp[j]);
            fprintf(fp,"%12.6lf", mp[0]);
            fprintf(fp,"\n");
            fclose(fp);
        }
        append = 1;
    }

/*  STEP 13. - Calculate extracellular potentials for given distance */
    append = append1 = append2 = 0;
    dist = RD;
    while ( dist < RD*10 ) {
        e[0] = 0.0;
        e[1] = y[1]*ecof[nh]*bess_k0(nu[nh]*dist);
        for ( j=1 ; j<nh ; j++ ) {
            e[2*j] = y[2*j]*ocof[j]*bess_k0(nu[j]*dist);
            e[2*j+1] = y[2*j+1]*ocof[j]*bess_k0(nu[j]*dist);
        }
        dist *= FAC;
        real_v(N, e, ep);

        if ( append ) {                         // extracellular potentials
            fp = fopen("extpot.dat","a");
            for ( j=0 ; j<N ; j++ ) fprintf(fp,"%12.6lf", ep[j]);
            fprintf(fp,"%12.6lf", ep[0]);
            fprintf(fp,"\n");
            fclose(fp);
        } else {
            fp = fopen("extpot.dat","w");
            for ( j=0 ; j<N ; j++ ) fprintf(fp,"%12.6lf", ep[j]);
            fprintf(fp,"%12.6lf", ep[0]);
            fprintf(fp,"\n");
            fclose(fp);
        }

        if ( append1 ) {
            fp = fopen("extdist.dat","a");
            for ( j=0 ; j<=N ; j++ ) fprintf(fp,"%12.6lf", dist);
            fprintf(fp,"\n");
            fclose(fp);
        } else {
            fp = fopen("extdist.dat","w");
            for ( j=0 ; j<=N ; j++ ) fprintf(fp,"%12.6lf", dist);
```

```
            fprintf(fp,"\n");
            fclose(fp);
        }

        if ( append2 ) {                              // z-values
            fp = fopen("extz.dat","a");
            for ( j=0 ; j<=N ; j++ ) fprintf(fp,"%9.4lf",x[j]);
            fprintf(fp,"\n");
            fclose(fp);
        } else {
            fp = fopen("extz.dat","w");
            for ( j=0 ; j<=N ; j++ ) fprintf(fp,"%9.4lf",x[j]);
            fprintf(fp,"\n");
            fclose(fp);
        }
        append = append1 = append2 = 1;
    }

/*  STEP 14. - Free vectors */
    free(mp);
    free(nu);
    free(icof);
    free(ecof);
    free(fac);
    free(c);
    free(cc);
    free(cur);
    free(e);
    free(ep);
    free(rad);
    free(y);
    return;
}

 /*************************************************************
                      OPERATION OF REAL_C and REAL_V
    REAL_C
    ------
    Enter with u_in[k] ( 0 <= k < N ) as the components of
    a real vector and exit with u_out[0]=c[0], u_out[1]=
    c[-N/2] and u_out[2k] + u_out_[2k+1]=c[k] ( 0 < k < N/2-1 ).

    REAL_V
    ------
    Enter with u_in[0]=c[0], u_in[1]=c[-N/2] and u_in[2k]+i
    u_in[2k+1] set to c[k] ( 0 < k < N/2-1 ) and exit with
    u_out[k]=u(x[k]) where ( 0 <= k < N-1 ).
  *************************************************************/
void fprime(double t, double *y, double *dy)
{
    extern double *fac;
    int k, kk, nh=N/2;
    double *v, tmp, h, m, n, ivdc;
    static double v_na=55.0, v_k=-72.0, v_l=-49.387, g_na=120.0, g_k=36.0, g_l=0.3;

/*  Step 1. - Create v[ ] and build Fourier coeffs of v[ ] */
    v = (double *) malloc( N*sizeof(double) );
    real_v( N, y, v);

/*  Step 2. - Build derivatives of H, M and N */
    for ( k=0 ; k<N ; k++ ) {
```

```
        h = y[N+k];
        m = y[2*N+k];
        n = y[3*N+k];
        ivdc = g_na*pow(m,3)*h*(v[k]-v_na)+g_k*pow(n,4)*(v[k]-v_k)+g_l*(v[k]-v_l);
        dy[k] = ivdc/CM;
        dy[N+k] = alfa_h(v[k])*(1.0-h)-beta_h(v[k])*h;
        dy[2*N+k] = alfa_m(v[k])*(1.0-m)-beta_m(v[k])*m;
        dy[3*N+k] = alfa_n(v[k])*(1.0-n)-beta_n(v[k])*n;
    }

/*  Step 3. - Compute Fourier coefficients of J_ivdc */
    real_c(N, dy, v);
    tmp = GA/(CM*RD);
    dy[0] = -v[0];
    dy[1] = -(tmp*y[1]*fac[nh]+v[1]);
    for ( k=1 ; k<nh ; k++ ) {
        kk = 2*k;
        dy[kk] = -(tmp*y[kk]*fac[k]+v[kk]);
        dy[kk+1] = -(tmp*y[kk+1]*fac[k]+v[kk+1]);
    }
    free(v);

    return;
}
```

A.1.2 HHDisperse.c

The second program, HHDisperse.c investigates the characteristics of a train of action potentials by manipulating the inter-spike interval. This problem is solved by means of a periodic solution, and therefore does not involve the initiation procedure necessary in the first problem. Instead this problem starts with the profile of a travelling wave which eventually settles to the travelling wave speed of the specified inter-spike interval.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/*****************************************************************
            Travelling Waves - Hodgkin Huxley - Squid Giant Axon

                SOLVES THE PERIODIC HODGKIN HUXLEY EQUATIONS
                    USING SPECTRAL METHOD - FOURIER

            C_m V_t + J = (g_A*R/2) V_xx  V(x,0) = F(x) given

  AND   (a) Spectral method - majority of program carried out in
            coefficent space - converting back for solution.
        (b) Calculates velocity for a range of wavelengths.
        (c) Three Dimensional Model - uses Bessel Functions.

  This program calculates & prints out:
        (i) wavelength and velocity                      dispvel.dat
       (ii) potential profiles                        Dispersion##
      (iii) Core current                                   Core##
       (iv) Radial current                                Radial##
        (v) Done file                                      Done##
*****************************************************************/

/*  Declaration of Global functions */
double bess_i0(double);
double bess_i1(double);
double bess_k0(double);
double bess_k1(double);
void fprime( double, double *, double *);
void sgsolve( int, double *, double *, double *, double, double *,
            void (*fcn)( double, double *, double *), int *);
void intrp( int, double, double *, double, double *, double *, int,double **, double *);
void step( int, double *, double *, double *, double *, double *, double *, int *,
           int *, int *, int *, double **, double *, double *, double *, double, double,
           void (*fcn)(double, double *, double *));
void real_c( int, double *, double *);
void real_v( int, double *, double *);
double wave( int, double);

/*  Biological Parameters */
#define        CELSIUS     18.5     /* Celsius temperature of neuron */
#define             RD    0.0238     /* Radius of squid axon (cm) */
#define             GA    28.249     /* Specific intra-cellular conductance (mmho/cm) */
#define             GE    28.249     /* Specific extra-cellular conductance (mmho/cm) */
#define             CM       1.0     /* Specific membrane capacitance (muF/cm^2) */
```

```
/*  Parameters for ODE solver */
#define              M       15     /* Node multiple */
#define           TEND     20.0     /* Final time */
#define             DT      0.5     /* Time step */
#define            TOL   5.e-16     /* Error tolerance */

/*  Declaration of HH coefficient functions */
double alfa_h( double );
double alfa_m( double );
double alfa_n( double );
double beta_h( double );
double beta_m( double );
double beta_n( double );

/*  Derivative of Hodgkin Huxley functions */
double d_alfa_h(double);
double d_alfa_m(double);
double d_alfa_n(double);
double d_beta_h(double);
double d_beta_m(double);
double d_beta_n(double);

double *fac;
int nodes;

void main(void)
{
    int k, FileNumber, pmax, qmax, rmax, pval, qval, rval, pstore, qstore, rstore, min,
        val, num;
    extern int nodes;
    double len, vel, dx;
    char filename[20], output[20], digit[4];
    FILE *fp;

    for ( FileNumber=1 ; FileNumber<121 ; FileNumber++ ) {
        num = FileNumber;
        digit[0] = num/100+48;
        num -= 100*(num/100);
        digit[1] = num/10+48;
        digit[2] = num%10+48;
        digit[3] = '\0';
        printf("\n File number is %s", digit);
        strcpy(filename,"done");
        strcat(filename,digit);
        if ( (fp=fopen(filename,"r")) == NULL ) {

/*  Step A - Fix wavelength (cm) and minimum number of nodes */
            len = 0.25*((double) FileNumber);
            nodes = M*FileNumber;

/*************************************************************
    Step B - Re-Estimate number of nodes.

    Suppose 2^p 3^q 5^r >= nodes then

        p*ln(2) + q*ln(3) + r*ln(5) >= ln(nodes)

    Thus 1 <= p <= pmax = ceil(ln(nodes)/ln(2))
         0 <= q <= qmax = ceil(ln(nodes)/ln(3))
         0 <= r <= rmax = ceil(ln(nodes)/ln(5))
```

```
    and choose p, q and r such that 2^p 3^q 5^r >= nodes
    taking the closest estimate >= nodes
*************************************************************/
            pmax = ((int) ceil(log((double) nodes)/log(2.0)));
            qmax = ((int) ceil(log((double) nodes)/log(3.0)));
            rmax = ((int) ceil(log((double) nodes)/log(5.0)));
            min = nodes;
            for ( pval=1; pval<=pmax ; pval++ ) {
                for ( qval=0 ; qval<=qmax ; qval++ ) {
                    for ( rval=0 ; rval<=rmax ; rval++ ) {
                        val = pow(2,pval)*pow(3,qval)*pow(5,rval);
                        if ( val >= nodes && val-nodes <= min ) {
                            pstore = pval;
                            qstore = qval;
                            rstore = rval;
                            min = val-nodes;
                        }
                    }
                }
            }
            nodes = pow(2,pstore)*pow(3,qstore)*pow(5,rstore);

/*  Step C - Fix wavelength (cm) and minimum number of nodes */
            printf("\nCalculating length %4.2lf cm with %5d nodes", len, nodes);
            vel = wave( FileNumber, len);
            strcpy(output,"DispVel");
            strcat(output, digit);
            fp = fopen(output,"w");
            fprintf(fp, "%2d\t %5d\t %4.2lf\t %8.6lf\n", FileNumber, nodes, len, vel);
            fclose(fp);
            fp = fopen(filename,"w");
            fclose(fp);
        } else {
            fclose(fp);
        }
    }
    return;
}

double wave(int FileNumber, double len )
{
    extern double *fac;
    extern double v[80], h[80], m[80], n[80];
    extern int nodes;
    double dx, *x, *y, *c, *cc, *cur, *mp, *nu, *icof, *ecof, ti, to, relerr, abserr,
           ivdc, vm, vr, tf, ts, xf, xs, tmp, pi, veq, meq, heq, neq, vn, vu, vl, vel;
    char filename[20], ext[4];
    int j, k, nh, ndim, ifail, im, start, num;
    FILE *fp;

 /*******************************************************
    Variables
    ---------
            y[0] -> y[N-1]      Holds the potential
            y[N] -> y[2*N-1]    Holds H at nodes
          y[2*N] -> y[3*N-1]    Holds M at nodes
          y[3*N] -> y[4*N-1]    Holds N at nodes
  *******************************************************/

/*  STEP 1. - Build ratio of Bessel functions */
    dx = len/((double) nodes);
```

```
    nh = nodes/2;
    pi = 4.0*atan(1.0);
    nu = (double *) malloc( (nh+1)*sizeof(double) );
    icof = (double *) malloc( (nh+1)*sizeof(double) );
    ecof = (double *) malloc( (nh+1)*sizeof(double) );
    fac = (double *) malloc( (nh+1)*sizeof(double) );
    cur = (double *) malloc( (nh+1)*sizeof(double) );
    fac[0] = cur[0] = nu[0] = 0.0;
    icof[0] = 1.0;
    ecof[0] = 0.0;
    for ( k=1 ; k<=nh ; k++ ) {
        tmp = 2.0*pi*RD*((double) k)/len;
        nu[k] = tmp;
        icof[k] = GE*bess_k1(tmp)/(GE*bess_k1(tmp)
                        *bess_i0(tmp)+GA*bess_k0(tmp)*bess_i1(tmp));
        ecof[k] = -GA*bess_i1(tmp)/(GE*bess_k1(tmp)
                        *bess_i0(tmp)+GA*bess_k0(tmp)*bess_i1(tmp));
        fac[k] = GE*tmp*bess_k1(tmp)*bess_i1(tmp)/(GE*bess_k1(tmp)
                        *bess_i0(tmp)+GA*bess_k0(tmp)*bess_i1(tmp));
        cur[k] = -GA*RD*GE*bess_k1(tmp)*bess_i1(tmp)/(GE*bess_k1(tmp)
                        *bess_i0(tmp)+GA*bess_k0(tmp)*bess_i1(tmp));
    }

/*  STEP 2. - Allocate memory to hold file information */
    start = 1;
    ndim = 4*nodes;
    x = (double *) malloc( (nodes+1)*sizeof(double) );
    y = (double *) malloc( ndim*sizeof(double) );
    mp = (double *) malloc( nodes*sizeof(double) );
    c = (double *) malloc( nodes*sizeof(double) );
    cc = (double *) malloc( nodes*sizeof(double) );

/*  STEP 3. - Initialise the membrane with the profile of a travelling wave */
    if ( (fp=fopen("InitialProfile.dat","r")) != NULL ) {
        for ( k=0 ; k<nodes ; k++ ) {
            x[k] = dx*((double) k);
            if ( fscanf ( fp,"%lf %lf %lf %lf", &mp[k], &y[nodes+k], &y[2*nodes+k],
                                                        &y[3*nodes+k]) == EOF ) {
                mp[k] = mp[k-1];
                y[nodes+k] = y[nodes+k-1];
                y[2*nodes+k] = y[2*nodes+k-1];
                y[3*nodes+k] = y[3*nodes+k-1];
            }
        }
        fclose(fp);
    } else {
        printf("\nCannot find input file!!");
        return(0.0);
    }
    x[nodes] = len;

/*  STEP 4. - Compute coefficients of initial voltage profile */
    real_c(nodes, mp, y);

/*  STEP 5. - Integrate forward in time */
    ti = 0.0;
    while ( ti < TEND ) {
        to = ti+DT;
        relerr = abserr = TOL;
        ifail = -1;
        while ( ti != to ) {
```

```
            sgsolve( ndim, &relerr, &abserr, &ti, to, y, fprimo, &ifail);
            if ( ifail != 2 ) printf("\nTrouble");
        }

/*  STEP 6. - Build voltage profile */
        real_v(nodes, y, mp);

/*  Step 7. - Calculate spike times */
        vm = mp[0];
        im = 0;
        for ( k=1 ; k<nodes ; k++ ) {
            if ( mp[k] > vm ) {
                im = k;
                vm = mp[k];
            }
        }
        printf("\nSpike potential %8.4lf", mp[im]);
        if ( im == 0 ) {
            vr = mp[1];
            vl = mp[nodes-1];
        } else if ( im == nodes-1 ) {
            vl = mp[nodes-2];
            vr = mp[0];
        } else {
            vl = mp[im-1];
            vr = mp[im+1];
        }

/*  Step 8. - Calculate conduction velocity */
        if ( start ) {
            xf = x[im]+0.5*dx*(vl-vr)/(vl-2.0*vm+vr);
            tf = ti;
            start = 0;
        } else {
            xs = xf;
            ts = tf;
            xf = x[im]+0.5*dx*(vl-vr)/(vl-2.0*vm+vr);
            tf = ti;
            vel = fabs(xf-xs);
            if ( len-vel < vel ) vel = len-vel;
            vel = vel/(tf-ts);
        }
        printf("\n Reached %4.2lf",ti);
    }

/*  STEP 9. - Output Information on Train of Action Potentials */
    num = FileNumber;
    ext[0] = num/100+48;
    num -= 100*(num/100);
    ext[1] = num/10+48;
    ext[2] = num%10+48;
    ext[3] = '\0';

/*  Step 9A. - Core current */
    c[0] = cur[0]*y[0];
    c[1] = -cur[nh]*y[1];
    for ( k=1 ; k<nh ; k++ ) {
        c[2*k] = -cur[k]*y[2*k+1];
        c[2*k+1] = cur[k]*y[2*k];
    }
    real_v(nodes, c, cc);
```

```
    strcpy(filename,"core");
    strcat(filename,ext);
    fp = fopen(filename,"w");
    for ( k=1 ; k<nh ; k++ ) fprintf(fp, "%15.10lf\t", cc[k]);
    fclose(fp);

/*  STEP 9B. - Radial current */
    c[0] = 0.0;
    c[1] = -2.0*pi*RD*GA*y[1];
    for ( k=1 ; k<nh ; k++ ) {
        c[2*k] = -2.0*pi*RD*GA*fac[k]*y[2*k+1];
        c[2*k+1] = -2.0*pi*RD*GA*fac[k]*y[2*k];
    }
    real_v(nodes, c, cc);
    strcpy(filename,"radial");
    strcat(filename,ext);
    fp = fopen(filename,"w");
    for ( k=1 ; k<nh ; k++ ) fprintf(fp, "%15.10lf\t", cc[k]);
    fclose(fp);

/*  STEP 9C. - Output file name */
    strcpy(filename,"Dispersion");
    strcat(filename,ext);
    fp = fopen(filename,"w");
    for ( k=0 ; k<nodes ; k++ ) {
        fprintf(fp,"%8.5lf\t %8.5lf\t %8.5lf\t %8.5lf\t %8.5lf\n",
                x[k], mp[k], y[nodes+k], y[2*nodes+k], y[3*nodes+k]);
        }
    fprintf(fp,"%8.5lf\t %8.5lf\t %8.5lf\t %8.5lf\t %8.5lf\n",
                  x[nodes], mp[0], y[nodes], y[2*nodes], y[3*nodes]);
    fclose(fp);

/*  STEP 12. - Free vectors */
    free(mp);
    free(nu);
    free(icof);
    free(ecof);
    free(fac);
    free(c);
    free(cc);
    free(cur);
    free(y);
    free(x);
    return vel;
}

/*************************************************************
                    OPERATION OF REAL_C and REAL_V
    REAL_C
    ------
    Enter with u_in[k] ( 0 <= k < N ) as the components of
    a real vector and exit with u_out[0]=c[0], u_out[1]=
    c[-N/2] and u_out[2k] + u_out_[2k+1]=c[k] ( 0 < k < N/2-1 ).

    REAL_V
    ------
    Enter with u_in[0]=c[0], u_in[1]=c[-N/2] and u_in[2k]+i
    u_in[2k+1] set to c[k] ( 0 < k < N/2-1 ) and exit with
    u_out[k]=u(x[k]) where ( 0 <= k < N-1 ).

*************************************************************/
```

```
void fprime(double t, double *y, double *dy)
{
    extern int nodes;
    extern double *fac;
    int k, kk, nh=nodes/2;
    double *v, tmp, h, m, n, ivdc;
    static double v_na=55.0, v_k=-72.0, v_l=-49.387,
                  g_na=120.0, g_k=36.0, g_l=0.3;

/*  Step 1. - Create v[ ] and build Fourier coeffs of v[ ] */
    v = (double *) malloc( nodes*sizeof(double) );
    real_v( nodes, y, v);

/*  Step 2. - Build derivatives of H, M and N */
    for ( k=0 ; k<nodes ; k++ ) {
        h = y[nodes+k];
        m = y[2*nodes+k];
        n = y[3*nodes+k];
        ivdc = g_na*pow(m,3)*h*(v[k]-v_na)+g_k*pow(n,4)*(v[k]-v_k)+g_l*(v[k]-v_l);
        dy[k] = ivdc/CM;
        dy[nodes+k] = alfa_h(v[k])*(1.0-h)-beta_h(v[k])*h;
        dy[2*nodes+k] = alfa_m(v[k])*(1.0-m)-beta_m(v[k])*m;
        dy[3*nodes+k] = alfa_n(v[k])*(1.0-n)-beta_n(v[k])*n;
    }

/*  Step 3. - Compute Fourier coefficients of J_ivdc */
    real_c(nodes, dy, v);
    tmp = GA/(CM*RD);
    dy[0] = -v[0];
    dy[1] = -(tmp*y[1]*fac[nh]+v[1]);
    for ( k=1 ; k<nh ; k++ ) {
        kk = 2*k;
        dy[kk] = -(tmp*y[kk]*fac[k]+v[kk]);
        dy[kk+1] = -(tmp*y[kk+1]*fac[k]+v[kk+1]);
    }
    free(v);
    return;
}
```

A.2 Extract data from Neurolucida file

As described in Chapter 3 the real morphological data needs to be carefully extracted from the Neurolucida data files; in particular, it is essential that the branching pattern of the neuron is maintained. This is achieved by defining structures in the program that are common to neuronal morphology, for example, branch, dendrite and synaptic contact. Each structure contains information about a particular object and can be defined an arbitrary number of times. For example, a branch structure contains the coordinates of an individual branch and its associated diameters at each of these points. This branch can be connected to other branch structures based on the notion of parent, child and peer discussed in Chapter 3. The result is in essence a dendrite, which is incorporated into the dendrite structure. Once the data has been extracted into the appropriate structures, minor calculations are required to find the length and surface area of each branch and dendrite, and the associated location of synaptic contacts within the dendritic tree. In the latter stages of this program there are small functions that take advantage of the recursive nature of the data management. It is a simple task to count contacts or terminal branches in the cell being investigated. The simulation of neuronal morphology uses this recursive methodology to build cells based on the original neuronal morphology extracted from the real cells.

The program BuildNeuron.c is the basic program necessary to extract the neuronal data into a form more suitable for analysis. The subsequent programs build on this foundation to solve two distinctly different problems, constructing an equivalent cable and simulating neuronal morphology.

A.2.1 BuildNeuron.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

typedef struct soma_t
{
/*  Physical properties of soma */
    int nobs;                   /* No. of observations in somal specification */

    double *x;                  /* X-coords of defining point */
    double *y;                  /* Y-coords of defining point */
    double *z;                  /* Z-coords of defining point */
    double *d;                  /* Diameter of soma at point (x,y,z) */
```

```
    double p_len;               /* Length of soma */

/*  Static biophysical properties of soma */
    double cs;                  /* Somal membrane capacitance (mu F/cm^2) */
    double ga;                  /* Intracellular conductance of soma (mS/cm) */
    double gs;                  /* Membrane conductance of soma (mS/cm^2) */

/*  Contact information */
    contact *conlist;           /* List of contacts */
    int ncon;
} soma;

typedef struct contact_t
{
    int id;

    double xc;                  /* X coordinate of contact */
    double yc;                  /* Y coordinate of contact */
    double zc;                  /* Z coordinate of contact */
    double dc;                  /* Dendritic diameter at contact */

    double xp;                  /* Projected X coordinate */
    double yp;                  /* Projected Y coordinate */
    double zp;                  /* Projected Z coordinate */

    double sd;                  /* Shortest distance to dendrite (micron) */
    double pl;                  /* Measurement of physical length (micron) */
    double el;                  /* Measurement of electrotonic length */
    double sa;                  /* Measurement of surface area (micron^2) */

    struct contact_t *prev;     /* Address of previous contact */
    struct contact_t *next;     /* Address of next contact */
} contact;

typedef struct branch_t
{
/*  Connectivity of branch */
    struct branch_t *parent;    /* Pointer to parent branch */
    struct branch_t *child;     /* Pointer to child branch */
    struct branch_t *peer;      /* Pointer to a peer branch */

/*  Physical properties of branch */
    int nobs;                   /* No. of observations in branch specification */
    double *x;                  /* X-coordinate of defining point */
    double *y;                  /* Y-coordinate of defining point */
    double *z;                  /* Z-coordinate of defining point */
    double *d;                  /* Diameter of dendrite */
    double p_len;               /* Length of branch */
    double *pl;                 /* Measurement of physical length (micron) */
    double *el;                 /* Measurement of electrotonic length */
    double *sa;                 /* Measurement of surface area (micron^2) */
    double e_len;               /* Total electrotonic length */

/*  Biophysical properties of branch */
    double cm;                  /* Dendritic membrane capacitance (mu F/cm^2) */
    double ga;                  /* Intracellular conductance (mS/cm) */
    double gm;                  /* Membrane conductance (mS/cm^2) */
    int bc;                     /* 0 - sealed */
                                /* 1 - cut */
                                /* 2 - leakage */
```

```
/*  Node information for spatial representation */
    int nodes;                  /* Total number nodes spanning branch */
    int jn;                     /* Junction node of the branch */
    int fn;                     /* Internal node connected to junction */
    double *c;                  /* Characteristic conductances */

/*  Contact information */
    contact *conlist;           /* List of contacts */
    int ncon;
} branch;

typedef struct dendrite_t
{
    branch *root;               /* Pointer to root branch of dendrite */
    double p_len;               /* Total length of dendrite */
    double area;                /* Total membrane area af dendrite */
} dendrite;

typedef struct neuron_t
{
    int ndend;                  /* Number of dendrites */
    dendrite *dendlist;         /* Pointer to an array of dendrites */
    soma *s;                    /* Soma structure */
} neuron;

/* Function type declarations */
neuron *Load_Sampled_Neuron(char *);
void Destroy_Sampled_Neuron(neuron *);
void init_branch( branch *, int, double, double, double);
void BuildContactInfo(contact *, branch *, branch **);
void remove_branch( branch **, branch *);
void build_dendrite( branch **, branch *);
void clean_dendrite( branch *);
void destroy_dendrite( branch *);
int count_branches( branch *, branch *);
int count_terminal_branches( branch *, branch *);
void branch_length( FILE *, branch *);
int count_branches( branch *, branch *);
int count_contacts( branch *, branch *);

/* Global definitions */
#define             CS    1.0
#define             GS    14.3
#define             GA    0.091
#define             CM    1.0
#define             GM    14.3

int main( int argc, char **argv)
{
    neuron *n;

/*  Load sampled neuron */
    if ( argc != 2 ) {
        printf("\n Invoke program with load <input>\n");
        return(1);
    } else {
        n = Load_Sampled_Neuron( argv[1] );
        if ( !n ) {
            printf("\n Failed to find sampled neuron\n");
            return(1);
        }
```

```
    }

/*  Clean up memory */
    Destroy_Sampled_Neuron( n );
    return(0);
}

neuron *Load_Sampled_Neuron(char *filename)
{
    int j, k, ncon, n, id, connected, ignored;
    double tmp, piby2, xold, xnew, yold, ynew, zold, znew, diam, xl, xr, dl, dr,
           px, py, pz, min;
    neuron *cell;
    soma *s;
    contact *oldcon, *newcon, *firstcon;
    branch *bold, *bnew, *first_branch, *bopt;
    char temp[100];
    FILE *input;

/*  STEP 1. - Open neuron data file */
    printf("\nOpening file %s\n",filename);
    if ( (input=fopen(filename,"r"))==NULL ) {
        printf("\nProblem loading neuron description from file\n");
        return NULL;
    }

/*  STEP 2. - Get memory for neuron structure */
    cell = (neuron *) malloc( sizeof(neuron) );
    s = cell->s = (soma *) malloc( sizeof(soma) );

/*  STEP 3, -  Initialise soma structure */
    s->x = (double *) malloc( n*sizeof(double) );
    s->y = (double *) malloc( n*sizeof(double) );
    s->z = (double *) malloc( n*sizeof(double) );
    s->d = (double *) malloc( n*sizeof(double) );
    s->gs = GS;
    s->ga = GA;
    s->cs = CS;
    s->conlist = NULL;
    s->ncon = 0;

/*  STEP 4. - Get soma morphological data */
    fscanf(input,"%lf %lf %lf %lf",&xold, &yold, &zold, &diam);
    s->x[0] = xold;
    s->y[0] = yold;
    s->z[0] = zold;
    s->d[0] = diam;
    for ( k=j=1,s->p_len=0.0 ; k<n ; k++ ) {
        fscanf(input,"%lf %lf %lf %lf",&xnew, &ynew, &znew, &diam);
        tmp = pow(xnew-xold,2)+pow(ynew-yold,2)+pow(znew-zold,2);
        if ( tmp > 0.01 ) {
            s->x[j] = xold = xnew;
            s->y[j] = yold = ynew;
            s->z[j] = zold = znew;
            s->d[j] = diam;
            j++;
            s->p_len += sqrt(tmp);
        }
    }
    s->nobs = j;
```

```
/*  STEP 5. - Get branch and contact data */
    oldcon = NULL;
    bold = NULL;
    fscanf(input,"%s", temp);
    do {
        if ( strcmp(temp, "Branch") == 0 ) {
            fscanf(input, "%d", &n);
            printf("Found a branch defined by %d nodes\n", n);
            bnew = (branch *) malloc( sizeof(branch) );
            if ( bold ) {
                bold->child = bnew;
            } else {
                first_branch = bnew;
            }
            bnew->parent = bold;
            bnew->peer = NULL;
            bnew->child = NULL;

/*  STEP 6. - Initialise branch */
            init_branch( bnew, n, CM, GM, GA);
            fscanf(input,"%lf %lf %lf %lf", &xold, &yold, &zold, &diam);
            bnew->x[0] = xold;
            bnew->y[0] = yold;
            bnew->z[0] = zold;
            bnew->d[0] = diam;
            for ( bnew->p_len=bnew->pl[0]=0.0,k=j=1 ; k<n ; k++ ) {
                fscanf(input,"%lf %lf %lf %lf",&xnew,&ynew,&znew,&diam);
                tmp=pow(xnew-xold,2)+pow(ynew-yold,2)+pow(znew-zold,2);
                if ( tmp > 0.01 ) {
                    bnew->p_len += sqrt(tmp);
                    bnew->pl[j] = bnew->p_len;
                    bnew->x[j] = xold = xnew;
                    bnew->y[j] = yold = ynew;
                    bnew->z[j] = zold = znew;
                    bnew->d[j] = diam;
                    j++;
                }
            }
            bnew->nobs = j;
            bold = bnew;
        } else if ( strcmp(temp, "Marker") == 0 ) {

/*  STEP 7. - Initialise marker */
            fscanf(input, "%d %d", &id, &n);
            for ( k=0 ; k<n ; k++ ) {
                newcon = (contact *) malloc( sizeof(contact) );
                newcon->sd = NULL;
                newcon->id = id;
                if ( oldcon ) {
                    oldcon->next = newcon;
                } else {
                    firstcon = newcon;
                }
                newcon->prev = oldcon;
                newcon->next = NULL;
                fscanf(input,"%lf %lf %lf %lf", &newcon->xc,
                    &newcon->yc, &newcon->zc, &newcon->dc );
                oldcon = newcon;
            }
        } else {
            printf("Unknown block type %s!\n", temp);
```

```
            return NULL;
        }
    } while  ( fscanf(input,"%s", temp)!=EOF );
    fclose(input);

/*  STEP 8. - Complete electrotonic lengths and surface area of branch */
    bold = first_branch;
    piby2 = 2.0*atan(1.0);
    while ( bold ) {
        bold->el[0] = 0.0;
        bold->sa[0] = 0.0;
        xl = bold->pl[0];
        dl = bold->d[0];
        tmp = 0.04*sqrt((bold->gm)/(bold->ga));
        for ( k=1 ; k<bold->nobs ; k++ ) {
            xr = bold->pl[k];
            dr = bold->d[k];
            bold->el[k] = bold->el[k-1]+tmp*(xr-xl)/(sqrt(dl)+sqrt(dr));
            bold->sa[k] = bold->sa[k-1]+piby2*(xr-xl)*(dl+dr);
            xl = xr;
            dl = dr;
        }
        bold->e_len = bold->el[bold->nobs-1];
        bold = bold->child;
    }

/*  STEP 9. - Associate contacts with branches and soma */
    ignored = 0;
    while ( firstcon ) {
        bold = first_branch;
        bopt = NULL;
        while ( bold ) {
            BuildContactInfo( firstcon, bold, &bopt);
            bold = bold->child;
        }
        newcon = firstcon->next;
        if ( firstcon->sd > 4.0 ) {

/*  STEP 9a. - Check for proximity to soma */
            px = firstcon->xc;
            py = firstcon->yc;
            pz = firstcon->zc;

/*  First stage is different from others */
            xnew = s->x[0]; ynew = s->y[0]; znew = s->z[0];
            firstcon->sd = min = sqrt(pow(xnew-px,2)
                          +pow(ynew-py,2)+pow(znew-pz,2))-(s->d[0]);

/*  Second stage compares points and projected points */
            for ( k=1 ; k<s->nobs ; k++ ) {
                xnew = s->x[k]; ynew = s->y[k]; znew = s->z[k];
                min = sqrt(pow(xnew-px,2)+pow(ynew-py,2)
                              + pow(znew-pz,2))-(s->d[k]);
                if ( min < firstcon->sd ) firstcon->sd = min;
            }
            if ( firstcon->sd < 4.0 ) {
                oldcon = s->conlist;
                if ( oldcon ) {
                    while ( oldcon->next ) oldcon = oldcon->next;
                    oldcon->next = firstcon;
                } else {
```

```
                    s->conlist = firstcon;
                }
                firstcon->prev = oldcon;
                firstcon->next = NULL;
                s->ncon++;
            } else {
                free(firstcon);
                ignored++;
            }
        } else {

/*  STEP 9b. - Check for proximity to soma */
            oldcon = bopt->conlist;
            if ( oldcon ) {
                while ( oldcon->next ) oldcon = oldcon->next;
                oldcon->next = firstcon;
            } else {
                bopt->conlist = firstcon;
            }
            firstcon->prev = oldcon;
            firstcon->next = NULL;
            bopt->ncon++;
        }
        firstcon = newcon;
    }

/*  STEP 10. - Count dendritic branches at soma */
    bold = first_branch;
    n = 0;
    while ( bold ) {
        bnew = first_branch;
        do {
            k = bnew->nobs-1;
            tmp = pow(bold->x[0]-bnew->x[k],2)+
                  pow(bold->y[0]-bnew->y[k],2)+
                  pow(bold->z[0]-bnew->z[k],2);
            connected = ( tmp < 0.01 );
            bnew = bnew->child;
        } while ( bnew && !connected );
        if ( !connected ) n++;
        bold = bold->child;
    }
    cell->ndend = n;

/* STEP 11. - Identify somal dendrites but extract nothing */
    cell->dendlist=(dendrite *) malloc((cell->ndend)*sizeof(dendrite));
    bold = first_branch;
    n = 0;
    while ( n < cell->ndend ) {
        bnew = first_branch;
        do {
            k = bnew->nobs-1;
            tmp = pow(bold->x[0]-bnew->x[k],2)+
                  pow(bold->y[0]-bnew->y[k],2)+
                  pow(bold->z[0]-bnew->z[k],2);
            connected = ( tmp < 0.01 );
            bnew = bnew->child;
        } while ( bnew && !connected );
        if ( !connected ) {
            cell->dendlist[n].root = bold;
            n++;
```

```
        }
        bold = bold->child;
    }

/*  STEP 13. - Extract root of each dendrite from dendrite list */
    for ( k=0 ; k<cell->ndend ; k++ ) {
        bold = cell->dendlist[k].root;
        remove_branch( &first_branch, bold);
    }

/*  STEP 14. - Build each dendrite from its root branch */
    for ( k=0 ; k<cell->ndend ; k++ ) {
        build_dendrite( &first_branch, cell->dendlist[k].root);
        clean_dendrite( cell->dendlist[k].root);
    }
    if ( first_branch ) printf("\nWarning: Unconnected branch segments still exist\n");
    return cell;
}

/*****************************************************************
                Function to initialise a BRANCH
*****************************************************************/
void init_branch( branch *b, int n, double cm, double gm, double ga)
{

/*  Allocate memory for spatial orientation of dendrite */
    b->x = (double *) malloc( n*sizeof(double) );
    b->y = (double *) malloc( n*sizeof(double) );
    b->z = (double *) malloc( n*sizeof(double) );

/*  Allocate memory for branch geometry */
    b->d = (double *) malloc( n*sizeof(double) );
    b->pl = (double *) malloc( n*sizeof(double) );
    b->el = (double *) malloc( n*sizeof(double) );
    b->sa = (double *) malloc( n*sizeof(double) );

/*  Set parameter values */
    b->cm = cm;
    b->gm = gm;
    b->ga = ga;

/*  Set boundary condition */
    b->bc = 0;

/*  Initialise node information */
    b->nodes = 0;
    b->fn = 0;
    b->jn = 0;
    b->c = NULL;

/*  Initialise contact information */
    b->conlist = NULL;
    b->ncon = 0;
    return;
}

/*****************************************************************
            Function to build CONTACT information
*****************************************************************/
void BuildContactInfo(contact *con, branch *b, branch **bopt)
{
```

```
    int k;
    double px, py, pz, tmp, xold, xnew, yold, ynew, zold, znew,
           numer, denom, xmin, ymin, zmin, min;

    px = con->xc;
    py = con->yc;
    pz = con->zc;

/*  STEP 1. - First stage is different from others */
    xnew = b->x[0]; ynew = b->y[0]; znew = b->z[0];
    min = sqrt(pow(xnew-px,2)+pow(ynew-py,2)+pow(znew-pz,2));
    if ( !(con->sd) || ( con->sd && min < con->sd ) ) {
        con->sd = min;
        con->xp = xnew; con->yp = ynew; con->zp = znew;
        con->pl = con->el = con->sa = 0.0;
        *bopt = b;
    }

/*  STEP 2. - Second stage compares points and projected points */
    for ( k=1 ; k<b->nobs ; k++ ) {
        xold = xnew; yold = ynew; zold = znew;
        xnew = b->x[k]; ynew = b->y[k]; znew = b->z[k];
        numer = (xnew-xold)*(px-xold)+(ynew-yold)*(py-yold)
                                     +(znew-zold)*(pz-zold);
        denom = pow(xnew-xold,2)+pow(ynew-yold,2)+pow(znew-zold,2);

/*  STEP 2a. - Project onto branch */
        if ( 0.0 <= numer && numer <= denom ) {
            tmp = numer/denom;
            xmin = (1.0-tmp)*xold+tmp*xnew;
            ymin = (1.0-tmp)*yold+tmp*ynew;
            zmin = (1.0-tmp)*zold+tmp*znew;
            min = sqrt(pow(xmin-px,2)+pow(ymin-py,2)+pow(zmin-pz,2));
            if ( !(con->sd) || ( con->sd && min < con->sd ) ) {
                con->sd = min;
                con->xp = xmin; con->yp = ymin; con->zp = zmin;
                con->pl = (1.0-tmp)*b->pl[k-1]+tmp*b->pl[k];
                con->el = (1.0-tmp)*b->el[k-1]+tmp*b->el[k];
                con->sa = (1.0-tmp)*b->sa[k-1]+tmp*b->sa[k];
                *bopt = b;
            }
        }

/*  STEP 2b. - Check proximity to points of branch */
        min = sqrt(pow(xnew-px,2)+pow(ynew-py,2)+pow(znew-pz,2));
        if ( !(con->sd) || ( con->sd && min < con->sd ) ) {
            con->sd = min;
            con->xp = xnew; con->yp = ynew; con->zp = znew;
            con->pl = b->pl[k]; con->el = b->el[k]; con->sa = b->sa[k];
            *bopt = b;
        }
    }
    return;
}

/*************************************************************
          Function to remove a branch from a branch list
*************************************************************/
void remove_branch(branch **head, branch *b)
{
    if ( !(*head) || !b ) return;
```

```
    if ( *head == b ) {
        *head = b->child;
        if( *head )  (*head)->parent = NULL;
    } else {
        b->parent->child = b->child;
        if ( b->child ) b->child->parent = b->parent;
    }
    b->parent = NULL;
    b->child = NULL;
    return;
}

/*************************************************************
         Function to build a dendrite from its root
*************************************************************/
void build_dendrite( branch **head, branch *root)
{
    int k;
    double tmp;
    branch *bnow, *bnext, *btmp;

    bnow = *head;
    while ( bnow ) {

/* Store bnow's child in case it's corrupted */
        bnext = bnow->child;

/* Search if proximal end of bnow is connected to distal end of root */
        k = (root->nobs)-1;
        tmp = pow(bnow->x[0]-root->x[k],2)
             +pow(bnow->y[0]-root->y[k],2)
             +pow(bnow->z[0]-root->z[k],2);
        if ( tmp <= 0.01 ) {

/* Take bnow out of the branch list */
            remove_branch( head, bnow);

/* Connect bnow to the root as the child or a peer of the child.
   Initialise childs' children and peers to NULL as default */
            bnow->child = NULL;
            bnow->peer = NULL;
            bnow->parent = root;

/* Inform root about its child if it's the first child, or add
   new child to first child's peer list */
            if ( root->child ) {
                btmp = root->child;
                while ( btmp->peer ) btmp = btmp->peer;
                btmp->peer = bnow;
            } else {
                root->child = bnow;
            }
        }

/* Initialise bnow to next branch in list */
        bnow = bnext;
    }

/* Iterate through remaining tree */
    if ( root->child ) build_dendrite( head, root->child);
    if ( root->peer ) build_dendrite( head, root->peer);
```

```
    return;
}

/*************************************************************
        Function to remove peerless children
*************************************************************/
void clean_dendrite( branch *root)
{
    int k, np, nc, mem, n;
    double tmp, sarea;
    contact *con;
    branch *btmp, *brem;

/*  Iterate through remaining tree */
    if ( root->child ) clean_dendrite( root->child );
    if ( root->peer ) clean_dendrite( root->peer );

/*  Extend original parent limb */
    brem = root->child;
    if ( brem && !(brem->peer) ) {
        root->child = brem->child;
        if ( brem->child ) {
            brem->child->parent = root;
            btmp = brem->child->peer;
            while ( btmp ) {
                btmp->parent = root;
                btmp = btmp->peer;
            }
        }
       root->bc = brem->bc;
//        root->nodes += (brem->nodes)-1;
        np = root->nobs;
        nc = brem->nobs;
        mem = np+nc-1;
        root->nobs = mem;
        root->x=(double *) realloc((void *)root->x,mem*sizeof(double));
        for ( k=np ; k<mem ; k++ ) root->x[k] = brem->x[k-np+1];
        root->y=(double *) realloc((void *)root->y,mem*sizeof(double));
        for ( k=np ; k<mem ; k++ ) root->y[k] = brem->y[k-np+1];
        root->z=(double *) realloc((void *)root->z,mem*sizeof(double));
        for ( k=np ; k<mem ; k++ ) root->z[k] = brem->z[k-np+1];
        root->d=(double *) realloc((void *)root->d,mem*sizeof(double));
        for ( k=np ; k<mem ; k++ ) root->d[k] = brem->d[k-np+1];
        root->pl=(double *) realloc((void *)root->pl,mem*sizeof(double));
        for (k=np ;k<mem ;k++ ) root->pl[k]=root->p_len+brem->pl[k-np+1];
        root->el=(double *) realloc((void *)root->el,mem*sizeof(double));
        for (k=np ;k<mem ;k++ ) root->el[k]=root->e_len+brem->el[k-np+1];
        sarea = root->sa[np-1];
        root->sa=(double *) realloc((void *)root->sa,mem*sizeof(double));
        for ( k=np ; k<mem ; k++ ) root->sa[k] = sarea+brem->sa[k-np+1];
        root->p_len += brem->p_len;
        root->e_len += brem->e_len;
        root->ncon += brem->ncon;
        con = root->conlist;
        if ( con ) {
            while ( con->next ) con = con->next;
            con->next = brem->conlist;
            if ( brem->conlist ) brem->conlist->prev = con;
        } else {
            root->conlist = brem->conlist;
        }
```

```
        brem->conlist = NULL;
        free(brem->x);
        free(brem->y);
        free(brem->z);
        free(brem->d);
        free(brem->pl);
        free(brem->el);
        free(brem->sa);
        if ( brem->c ) free(brem->c);
        free ( brem );
    }
    return;
}

/*************************************************************
                Function to destroy a NEURON
*************************************************************/
void Destroy_Sampled_Neuron(neuron *cell)
{
    int i;
    contact *prevcon, *nextcon;

/*  Free Soma */
    free ( cell->s->x );
    free ( cell->s->y );
    free ( cell->s->z );
    free ( cell->s->d );
    prevcon = cell->s->conlist;
    while ( prevcon ) {
        nextcon = prevcon->next;
        free ( prevcon );
        prevcon = nextcon;
    }
    free ( cell->s );

    for (i=0;i<cell->ndend;i++) destroy_dendrite(cell->dendlist[i].root);
    free(cell);
    return;
}

/*************************************************************
                Function to destroy DENDRITE
*************************************************************/
void destroy_dendrite( branch *b )
{
    int i;
    contact *prevcon, *nextcon;

    if ( b->child ) destroy_dendrite(b->child);
    if ( b->peer ) destroy_dendrite(b->peer);
    free(b->x);
    free(b->y);
    free(b->z);
    free(b->d);
    free(b->pl);
    free(b->el);
    free(b->sa);
    if ( b->c ) free(b->c);
    prevcon = b->conlist;
    while ( prevcon ) {
        nextcon = prevcon->next;
```

```
        free ( prevcon );
        prevcon = nextcon;
    }
    free ( b );
    return;
}

/*************************************************************
        Function to count contacts from current branch
                    to the dendritic tip.
*************************************************************/
int count_contacts( branch *bstart, branch *bnow)
{
    static int n;
    contact *con;

    if ( bstart == bnow ) n = 0;
    if ( bnow ) {
        if ( bnow->child ) count_contacts(bstart, bnow->child);
        if ( bnow->peer ) count_contacts(bstart, bnow->peer);
        con = bnow->conlist;
        while ( con ) {
            n++;
            if ( con->sd > 4.0 )
                printf("\nContact not close to dendrite %6.2lf", con->sd);
            con = con->next;
        }
    }
    return n;
}

/*************************************************************
             Function to count number of branches
*************************************************************/
int count_branches( branch *bstart, branch *bnow)
{
    static int n;

    if ( bstart == bnow ) n = 0;
    if ( bnow ) {
        if ( bnow->child ) count_branches(bstart, bnow->child);
        if ( bnow->peer ) count_branches(bstart, bnow->peer);
        n++;
    }
    return n;
}

/*************************************************************
             Function to find length of dendrite from
                    current branch to tips.
*************************************************************/
double branch_length( branch *bstart, branch *bnow)
{
    static double length;

    if ( bstart == bnow ) length = 0.0;
    if ( bnow ) {
        if ( bnow->child ) branch_length(bstart, bnow->child);
        if ( bnow->peer ) branch_length(bstart, bnow->peer);
        length += bnow->p_len;
    }
```

```
    return length;
}

/*************************************************************
            Function to count number of terminal branches
*************************************************************/
int count_terminal_branches( branch *bstart, branch *bnow)
{
    static int n;

    if ( bstart == bnow ) n = 0;
    if ( bnow ) {
        if ( bnow->child ) count_terminal_branches(bstart, bnow->child);
        if ( bnow->peer ) count_terminal_branches(bstart, bnow->peer);
        if ( !bnow->child ) n++;
    }
    return n;
}
```

A.3 Equivalent cables

The program MapContacts.c uses the BuildNeuron.c program as a foundation to extract morphological and synaptic neuronal data for the construction of equivalent cables. MapContacts.c uses the transformation procedures described in Chapter 4 to construct an equivalent cable with the associated bijective mapping of input.

A.3.1 MapContacts.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/* Function type declarations */
neuron *Load_Sampled_Neuron(char *);
void Destroy_Sampled_Neuron(neuron *);
void init_branch( branch *, int, double, double, double);
void BuildContactInfo(contact *, branch *, branch **);
void remove_branch( branch **, branch *);
void build_dendrite( branch **, branch *);
void clean_dendrite( branch *);
void destroy_dendrite( branch *);
int count_branches( branch *, branch *);
double BranchPhysicalLength( branch *, branch *);
double BranchElectrotonicLength( branch *, branch *);
int BuildElectrotonicNodes( branch *, double, int, int);
void ConstructTreeMatrix( branch *, double **, double *);
int count_contacts( branch *, branch *);
int count_terminal_branches( branch *, branch *);
void OutputProperties( branch * );
void house( int, double **, double *, double *);
void MapContacts( branch *, double *, int);

/* Global definitions */
#define             CS      1.0
#define             GS      14.3
#define             GA      14.3
#define             CM      1.0
#define             GM      0.091
#define             EU      0.2
#define             ID      1
#define        OUTPUT2      "curr_cell.out"
#define        OUTPUT1      "cable_cell.out"

int main( int argc, char **argv)
{
    int j, k, id, start, nodes, nc, fn, num;
    double elen, csum, tmp, fac, pi, theta, **a, *curr, *amp, *d, *dtree,*s, *stree, *e;
    neuron *n;
    branch *bran;
    FILE *fp;

/*  Load sampled neuron */
    if ( argc != 2 ) {
```

```
        printf("\n Invoke program with load <input>\n");
        return(1);
    } else {
        n = Load_Sampled_Neuron( argv[1] );
        if ( !n ) {
            printf("\n Failed to find sampled neuron\n");
            return(1);
        }
    }

/*  Count contacts */
    pi = 4.0*atan(1.0);
    for ( nc=k=0 ; k<n->ndend ; k++ ) {
        nc += count_contacts( n->dendlist[k].root, n->dendlist[k].root);
    }
    printf("\n Located %d contacts on dendrites", nc);
    printf("\n       Located %d contacts on soma", n->s->ncon);

/*  Recompute dendritic diameters */
    for ( k=0 ; k<n->ndend ; k++ ) OutputProperties( n->dendlist[k].root);

/*  Compute entire electrotonic length of a neuron */
    for (elen=0.0,k=0;k<n->ndend;k++) elen += BranchElectrotonicLength
                            ( n->dendlist[k].root, n->dendlist[k].root);
    printf("\nTotal Electrotonic Length is %12.6lf", elen);

/*  STAGE 1. - Discretise electrotonic neuron */
    fn = 1;
    for ( k=0 ; k<n->ndend ; k++ ) {
        fn = BuildElectrotonicNodes( n->dendlist[k].root, EU, 0, fn);
    }
    printf("\n No nodes is %d", fn);

/*  STAGE 2. - Build the model matrices */
    e = (double *) malloc( fn*sizeof(double) );
    d = (double *) malloc( fn*sizeof(double) );
    dtree = (double *) malloc( fn*sizeof(double) );
    curr = (double *) malloc( fn*sizeof(double) );
    a = (double **) malloc( fn*sizeof(double *) );
    for ( k=0 ; k<fn ; k++ ) {
        d[k] = 0.0;
        dtree[k] = 0.0;
        curr[k] = 0.0;
        a[k] = (double *) malloc( fn*sizeof(double) );
        for ( j=0 ; j<fn ; j++ ) a[k][j] = 0.0;
    }

/*  STAGE 3. - Do soma node */
    for ( csum=0.0,k=0 ; k<n->ndend ; k++ ) csum += (n->dendlist[k].root)->c[0];
    dtree[0] = csum;
    a[0][0] = -1.0;
    for ( k=0 ; k<n->ndend ; k++ ) {
        bran = n->dendlist[k].root;
        j = bran->fn;
        a[0][j] = (bran->c[0])/csum;
    }
    for ( k=0 ; k<n->ndend ; k++ ) ConstructTreeMatrix( n->dendlist[k].root, a, dtree);

/*  STAGE 3a. - Consistency check */
    for ( j=0 ; j<fn ; j++ ) {
        for ( k=0 ; k<fn ; k++ ) {
```

```
            if ( a[j][k] == 0.0 && a[k][j] != 0.0 ) printf("\n Trouble! %d %d", j, k);
        }
    }

/*  STAGE 4. - Symmetrise the tree matrix */
    stree = (double *) malloc( fn*sizeof(double) );
    for ( k=0 ; k<fn ; k++ ) stree[k] = 0.0;
    stree[0] = 1.0;
    for ( j=0 ; j<fn ; j++ ) {
        for ( k=0 ; k<fn ; k++ ) {
            if ( a[j][k] != 0.0 && stree[j] != 0.0 )
                        stree[k] = stree[j]*sqrt(a[k][j]/a[j][k]);
        }
    }
    for (k=0;k<fn;k++) if (!stree[k]) printf("\n Entry %d is zero", k);

/*  STAGE 5. - Build the symmetrised tree matrix */
    for ( j=0 ; j<fn ; j++ ) {
        for ( k=j+1 ; k<fn ; k++ ) {
            if ( a[j][k] != 0.0 ) a[j][k]=a[k][j]=sqrt(a[j][k]*a[k][j]);
        }
    }

/*  STAGE 6. - Apply the Householder procedure */
    house( fn, a, d, e);

/*  STAGE 7. - Construct the equivalent cable */
    for ( d[1]=0.0,k=0 ; k<n->ndend ; k++) d[1] += n->dendlist[k].root->c[0];
    theta = 0.5*pi;
    nc = 1;
    while( nc < fn-1 && fabs(theta) > 0.01 ) {
        fac = fabs(e[nc])/sin(theta);
        if ( fabs(fac) <= 1.0 ) {
            theta = acos(fac);
            d[nc+1] = d[nc]*pow(tan(theta),2);
            nc++;
        } else {
            theta = 0.0;
        }
    }

/*  STAGE 8. - Extract physical dimension: c=(pi/2)sqrt(g_m*g_a)d^{3/2} */
    fac = 2.0/(pi*sqrt(GM*GA));
    fp = fopen(OUTPUT1,"w");
    for (k=1 ; k<=nc ; k++)fprintf(fp,"(%3d,%5.2lf),",k,-1.0e4*pow(d[k]*fac,0.6666667));
    fprintf(fp,"\n\n");
    for (k=1 ; k<=nc ; k++)fprintf(fp,"(%3d,%5.2lf),",k,-1.0e4*pow(d[k]*fac,0.6666667));
    fclose(fp);

/*  STAGE 9. - Construct vector of current inputs */
    amp = (double *) malloc( fn*sizeof(double) );
    for ( k=0 ; k<fn ; k++ ) amp[k] = 0.0;
    for ( k=0 ; k<n->ndend ; k++ ) MapContacts(n->dendlist[k].root, amp, ID );

/*  STAGE 10a. - Construct symmetrising diagonal matrix S */
    s = (double *) malloc( (nc+1)*sizeof(double) );
    s[0] = 1.0;
    d[0] = 0.0;
    for ( k=0 ; k<nc-1 ; k++ ) {
        tmp = (d[k]+d[k+1])/(d[k+1]+d[k+2]);
        s[k+1] = s[k]*sqrt(tmp);
```

```
    }
    tmp = d[nc-1]/d[nc]+1.0;
    s[nc] = s[nc-1]*sqrt(tmp);

/*  STAGE 10b. - Correct for negatives */
    for ( k=0 ; k<=nc ; k++ ) if ( e[k] < 0.0 ) s[k] = -s[k];

/*  STAGE 10c. - Construct Current EGP Matrix */
    for ( k=0 ; k<fn ; k++ ) dtree[k] *= stree[k];
    for ( k=0 ; k<fn ; k++ ) {
        for ( j=0 ; j<fn ; j++ ) a[k][j] /= dtree[j];
    }
    for ( k=0 ; k<=nc ; k++ ) d[k] *= s[k];
    for ( k=0 ; k<=nc ; k++ ) {
        for ( j=0 ; j<fn ; j++ ) a[k][j] *= d[k];
    }

/*  STAGE 11. - Calculate injected current on cable */
    for ( k=0 ; k<=nc ; k++ ) {
        for ( j=0 ; j<fn ; j++ ) curr[k] += a[k][j]*amp[j];
    }
    fp = fopen(OUTPUT2,"w");
    for ( k=0 ; k<nc ; k++ ) fprintf(fp,"%4.16lf\n",curr[k]);
    fclose(fp);

/*  Clean up memory */
    for ( k=0 ; k<fn ; k++ ) free(a[k]);
    free(a);
    free(amp);
    free(curr);
    free(d);
    free(dtree);
    free(e);
    free(s);
    free(stree);
    Destroy_Sampled_Neuron( n );
    return(0);
}

/*************************************************************
  Performs Householder transformations on a symmetric matrix.
*************************************************************/
void house( int n, double **a, double *d, double *e)
{
    int i, j, k;
    double beta, g, s, sum, *q, *u, *w;

/*  Allocate two working vectors each of length n */
    q = (double *) malloc( n*sizeof(double) );
    u = (double *) malloc( n*sizeof(double) );
    w = (double *) malloc( n*sizeof(double) );

/*  A total of (n-2) householder steps are required - start on the
    first row of a[ ][ ] and progress to the third last row - the
    last 2 rows already conform to the tri-diagonal structure */
    e[0] = 0.0;
    for ( i=0 ; i<n-2 ; i++ ) {
        d[i] = a[i][i];

/*  Determine the magnitude of the working row */
        for ( s=0.0,j=i+1 ; j<n ; j++ ) s += a[i][j]*a[i][j];
```

```
        s = sqrt(s);
        if ( a[i][i+1] < 0.0 ) s = -s;
        e[i+1] = -s;
        g = s*a[i][i+1];
        if ( s == 0.0 ) {
            a[i][i] = 1.0;
        } else {
            beta = 1.0/(s*g);
            u[i+1] = g;
            for ( j=i+2 ; j<n ; j++ ) u[j] = a[i][j];
            for ( j=i+1 ; j<n ; j++ ) {
                for ( sum=0.0,k=i+1 ; k<n ; k++ ) sum += a[j][k]*u[k];
                w[j] = sum*beta;
            }
            for ( sum=0.0,j=i+1 ; j<n ; j++ ) sum += u[j]*w[j];
            sum *= 0.5*beta;
            for ( j=i+1 ; j<n ; j++ ) q[j] = w[j]-sum*u[j];
            for ( j=i+1 ; j<n ; j++ ) {
                for (k=i+1;k<n;k++) a[j][k]-=(q[j]*u[k]+u[j]*q[k]);
            }

/*  Store vector to generate orthogonal matrix */
            a[i][i] = beta;
            for ( j=i+1 ; j<n ; j++ ) a[i][j] = u[j];
        }
    }
    d[n-2] = a[n-2][n-2];
    d[n-1] = a[n-1][n-1];
    e[n-1] = a[n-2][n-1];

/*  Restructure a[ ][ ] to hold product of Householder matrices */
    a[n-2][n-1] = a[n-1][n-2] = 0.0;
    a[n-2][n-2] = a[n-1][n-1] = 1.0;
    for ( i=n-3 ; i>=0 ; i-- ) {
        beta = a[i][i];
        for ( j=i+1 ; j<n ; j++ ) u[j] = a[i][j];
        a[i][i] = 1.0;
        for ( j=i+1 ; j<n ; j++ ) a[i][j] = 0.0;
        for ( j=i+1 ; j<n ; j++ ) a[j][i] = 0.0;
        for ( j=i+1 ; j<n ; j++ ) {
            for ( sum=0.0,k=i+1 ; k<n ; k++ ) sum += a[j][k]*u[k];
            w[j] = sum*beta;
        }
        for ( j=i+1 ; j<n ; j++ ) {
            for ( k=i+1 ; k<n ; k++ ) a[j][k] -= u[j]*w[k];
        }
    }
    free(q);
    free(u);
    free(w);
    return;
}

/*************************************************************************
    Counts nodes required to discretise the electrotonic neuron.

    b  - Dendritic branch
    ql - Quantum of electrotonic length
    jn - Node number of branch point, the proximal node of a branch
    fn - Node number assigned to interior node adjacent to proximal node
*************************************************************************/
```

```
int BuildElectrotonicNodes( branch *b, double ql, int jn, int fn)
{
    int j, k, nodes;
    static int total_nodes;
    double dval, ElectrotonicStepsize;

/*  STEP 1. - Initialise counter if b is a root dendrite */
    if ( b->parent == NULL ) total_nodes = fn;

/*  STEP 2. - Compute required number of nodes */
    nodes = b->e_len/ql;
    if ( fmod(b->e_len/ql,1.0) > 0.5 ) nodes++;
    nodes++;

/*  STEP 3. - Set junction node, first node and node counter */
    b->nodes = nodes;
    b->jn = jn;
    b->fn = fn;
    total_nodes += (nodes-1);

/*  STEP 4. - Create vector of characteristic conductances for branch */
    ElectrotonicStepsize = b->e_len/((double) nodes-1);
    b->c = (double *) malloc( (nodes-1)*sizeof(double) );
    b->c[0] = 0.0;
    for ( j=1 ; j<nodes-1 ; j++ ) {
        dval = ElectrotonicStepsize*((double) j);
        k = 0;
        while ( k < b->nobs-1 && dval > b->el[k] ) k++;
        b->c[j] = b->sa[k-1]+(b->sa[k]-b->sa[k-1])*
                          (dval-(b->el[k-1]))/(b->el[k]-b->el[k-1]);
    }
    for ( k=0 ; k<nodes-2 ; k++ ) b->c[k] = b->c[k+1]-b->c[k];
    b->c[nodes-2] = b->sa[b->nobs-1]-b->c[nodes-2];
    for ( k=0 ; k<nodes-1 ; k++ ) b->c[k] *= (b->gm/ElectrotonicStepsize);

/*  STEP 5. - Iterate */
    if (b->child) BuildElectrotonicNodes
                                (b->child,ql,total_nodes-1,total_nodes);
    if (b->peer) BuildElectrotonicNodes( b->peer, ql, jn, total_nodes);
    return total_nodes;
}

/*************************************************************
            Construct tree matrices for equivalent cable
*************************************************************/
void ConstructTreeMatrix( branch *b, double **a, double *d)
{
    int j, k, row, node;
    branch *bran;
    double csum, tmp;

/*  STEP 1. - Do internal nodes */
    row = b->fn;
    node = 1;
    while ( node < b->nodes-1 ) {
        csum = b->c[node-1]+b->c[node];
        d[row] = csum;
        if ( node == 1 ) {
            a[row][b->jn] = b->c[node-1]/csum;
        } else {
            a[row][row-1] = b->c[node-1]/csum;
```

```
        }
        a[row][row] = -1.0;
        a[row][row+1] = b->c[node]/csum;
        row++;
        node++;
    }

/*  STEP 2. - Do branch point */
    row = b->fn+b->nodes-2;
    csum = b->c[b->nodes-2];
    if ( b->child ) {
        bran = b->child;
        csum += bran->c[0];
        while ( bran->peer ) {
            bran = bran->peer;
            csum += bran->c[0];
        }
    }
    d[row] = csum;
    if ( node == 1 ) {
        a[row][b->jn] = b->c[b->nodes-2]/csum;
    } else {
        a[row][row-1] = b->c[b->nodes-2]/csum;
    }
    a[row][row] = -1.0;
    if ( b->child ) {
        bran = b->child;
        a[row][bran->fn] = bran->c[0]/csum;
        while ( bran->peer ) {
            bran = bran->peer;
            a[row][bran->fn] = bran->c[0]/csum;
        }
    }

/*  STEP 3. - Iterate */
    if ( b->child ) ConstructTreeMatrix ( b->child, a, d );
    if ( b->peer ) ConstructTreeMatrix( b->peer, a, d );
    return;
}

/*****************************************************************
        Map contacts onto Equivalent Cable - type 1
*****************************************************************/
void MapContacts( branch *b, double *amp, int id)
{
    int j, k;
    double ess, frac, tmp;
    contact *con;

/*  Step 1. - Iterate through tree */
    if ( b->child ) MapContacts( b->child, amp, id );
    if ( b->peer ) MapContacts( b->peer, amp, id );

/*  Step 2. - Allocate currents */
    ess = (b->e_len)/((double) b->nodes-1);
    con = b->conlist;
    while ( con ) {
        if ( con->id == id ) {
            tmp = con->el/ess;
            j = ((int) floor(tmp));
            frac = fmod(tmp,1.0);
```

```
            if ( j == 0 ) {

/*  Step 2a. - One node is at the junction */
                amp[b->jn] += 1.0-frac;
                amp[b->fn] += frac;
            } else {

/*  Step 2b. - Both nodes are internal */
                amp[b->fn+j-1] += 1.0-frac;
                amp[b->fn+j] += frac;
            }
        }
        con = con->next;
    }
    return;
}

/*************************************************************
             Function to find electrotonic length of a
               dendrite from current branch to tips.
*************************************************************/
double BranchElectrotonicLength( branch *bstart, branch *bnow)
{
    static double elen;

    if ( bstart == bnow ) elen = 0.0;
    if ( bnow ) {
        if ( bnow->child ) BranchElectrotonicLength(bstart, bnow->child);
        if ( bnow->peer ) BranchElectrotonicLength(bstart, bnow->peer);
        elen += bnow->e_len;
        printf("\n%12.6lf \t %12.6lf \t %12.6lf \t %12.6lf",
            bnow->e_len, bnow->p_len, bnow->d[0], bnow->d[bnow->nobs-1]);
    }
    return elen;
}

/*************************************************************
                Function to output branch diameters
*************************************************************/
void OutputProperties( branch *b )
{
    int i, k;
    static int start=1;
    double dold, dnew, len, xold, yold, zold, xnew,
           ynew, znew, dx, dy, dz, size;
    branch *bran;
    FILE *fp;

    if ( b->child ) OutputProperties(b->child);
    if ( b->peer ) OutputProperties(b->peer);
    if ( start ) {
        fp = fopen("output","w");
        start = 0;
    } else {
        fp = fopen("output","a");
        fprintf(fp,"\n");
    }

/*  Decomposes branches into lengths of uniform diameter */
    len = xold = b->pl[1];
    dold = b->d[1];
```

```
    for ( k=2 ; k<b->nobs ; k++ ) {
        xnew = b->pl[k];
        dnew = b->d[k];
        if ( dnew != dold ) {
            len += 0.5*(xnew-xold);
            fprintf(fp,"%6.2lf \t %6.2lf \n", len, dold);
            len = 0.5*(xnew-xold);
        } else {
            len += xnew-xold;
        }
        xold = xnew;
        dold = dnew;
    }
    fprintf(fp,"%6.2lf \t %6.2lf \n", len, dold);

/*  Constructs diameters of a branch and its children/peers */
    if ( b->child ) {
        fprintf(fp,"%6.2lf\t%6.2lf\t",b->d[(b->nobs)-1],b->child->d[1]);
        bran = b->child;
        while ( bran->peer ) {
            bran = bran->peer;
            fprintf(fp,"%6.2lf \t", bran->d[1]);
        }
    }

/*  Prints out branch lengths */
    printf("\nBranch length %6.2lf, %6.2lf, %6.2lf", b->p_len,b->d[0],b->d[b->nobs-1] );
    fclose(fp);
    return;
}
```

A.3.2 CumulativeCurrent.c

The distribution of current calculated by MapContacts.c is the total current at each node on the cable. The cumulative current density was constructed to visualise the distribution of current along the cable using the program CumulativeCurrent.c below.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/*****************************************************************
       Constructs the cumulative current input
*****************************************************************/

#define     EU        0.001     /*  Electrotonic units */
#define     EL           10     /*  Electrotonic length of intervals */

int main( int argc, char **argv)
{
    int j, k, span, n, nc, left, start;
    double *aver, eu, *sum, tmp, *curr;
    char filename[80], output1[80], *pnt, name[80];
    FILE *fp, *fp1;

/*  Load current cells */
    if ( argc != 2 ) {
        printf("\n Invoke program with load <input>\n");
        return(1);
    }
    if ( (fp1=fopen(argv[1],"r")) == NULL ) {
        printf("\n No file call %s\n",argv[1]);
        return(1);
    }

    while ( fscanf(fp1,"%s", &filename) != EOF ) {

/*  STEP 1. - Load file of current data */
        n = 0;
        if ( (fp=fopen(filename,"r"))!=NULL ) {
            while ( fscanf(fp, "%lf %lf", &eu, &tmp)!=EOF ) n++;
            fclose(fp);

/*  STEP 2. - Allocate memory to hold file information */
            curr = (double *) malloc( n*sizeof(double) );
            fp = fopen(filename,"r");
            for ( k=0 ; k<n ; k++ ) fscanf(fp, "%lf %lf", &eu, &curr[k]);
            fclose(fp);
        } else {
            printf("\nInput file not found\n");
            return(0);
        }
        printf("%s\t %d items in file\n", filename, n);

/*  STEP 3. - Averaging process */
        span = n/EL;
        left = n%EL;
        sum = (double *) malloc( (span)*sizeof(double) );
```

```
        for ( k=0 ; k<span ; k++ ) sum[k] = 0.0;
        printf("\n%d intervals\n%d nodes not included", span, left);

        for ( k=0 ; k<span ; k++ ) {
            for ( j=0 ; j<EL*(k+1) ; j++ ) sum[k] += curr[j];
        }

/*  STEP 4. - Create individual files */
        pnt = strstr( filename,"cell");
        k = 0;
        while ( *pnt != '.' ) {
            name[k++] = *pnt;
            pnt++;
        }
        name[k++] = '_';
        name[k++] = 's';
        name[k++] = 'u';
        name[k++] = 'm';
        name[k++] = '.';
        name[k++] = 'o';
        name[k++] = 'u';
        name[k++] = 't';
        name[k] = '\0';

        fp=fopen(name,"w");
        for ( k=0 ; k<span ; k++ ) fprintf(fp, "%lf\n", sum[k]);
        fclose(fp);

/*  Clean up memory */
        free(sum);
        free(curr);
    }
    fclose(fp1);
    return(0);
}
```

A.3.3 Lagrange.c

To allow comparison between the cumulative current input from different types of synaptic input, the curves were smoothed and then normalised. The program Lagrange.c performs both of these actions, with the additional calculation of the current density.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/*****************************************************************
            Program to filter the cumulative curves of cable
                using Lagrange method - normalise curves
*****************************************************************/

#define     EU        0.001     /*  Electrotonic units */
#define     EL           10     /*  Electrotonic length of intervals */
#define      N            4     /*  No. of polynomials */
#define      M            3     /*  No. of constraints */

/* Function type declaration */
ludcmp( int, double **, int *);
void linsol( int, double **, double *, double *, int *);

int main( int argc, char **argv)
{
    int m, i, j, k, node, nobs, nc, jj, *row;
    double alt, *a, *b, **c, *x, *t, tmp, len, pi, te,
           to, theta, xmax, diff, angle;
    char filename[80], output1[80], *ptr, name[80];
    FILE *fp, *fp1;

/*  Load current cells */
    if ( argc != 2 ) {
        printf("\n Invoke program with load <input>\n");
        return(1);
    }
    if ( (fp1=fopen(argv[1],"r")) == NULL ) {
        printf("\n No file call %s\n",argv[1]);
        return(1);
    }
    while ( fscanf(fp1,"%s", &filename) != EOF ) {

/*  STEP 1. - Load file of current data */
        nobs = 0;
        if ( (fp=fopen(filename,"r"))!=NULL ) {
            while ( fscanf(fp, "%lf", &tmp)!=EOF ) nobs++;
            fclose(fp);

/*  STEP 2. - Allocate memory to hold file information */
            x = (double *) malloc( (nobs+1)*sizeof(double) );
            t = (double *) malloc( (N+1)*sizeof(double) );
            a = (double *) malloc( (N+M+1)*sizeof(double) );
            b = (double *) malloc( (N+M+1)*sizeof(double) );
            c = (double **) malloc( (N+M+1)*sizeof(double *) );
            for ( k=0 ; k<(N+M+1) ; k++ ) {
                b[k] = 0.0;
```

```
            c[k] = (double *) malloc( (N+M+1)*sizeof(double) );
            for ( j=0 ; j<(N+M+1) ; j++ ) c[k][j] = 0.0;
        }
        fp = fopen(filename,"r");
        x[0] = 0.0;
        for ( k=1 ; k<=nobs ; k++ ) fscanf(fp, "%lf", &x[k]);
        xmax = x[nobs];
        fclose(fp);
    } else {
        printf("\nInput file not found\n");
        return 0;
    }
    printf("%s\t %d items in file\n", filename, nobs);

/*  STEP 3. - Lagrange Multipliers */
    len = ((double) nobs);
    for ( k=0 ; k<=nobs ; k++ )  {
        theta = 2.0*acos(sqrt(((double) k)/len));
        for ( j=0 ; j<=N ; j++ ) t[j] = cos(theta*((double) j));
        for ( j=0 ; j<=N ; j++ ) {
            tmp = t[j];
            for ( m=0 ; m<=N ; m++ ) c[j][m] += tmp*t[m];
            b[j] += tmp*x[k];
        }
    }

/*  STEP 3a. - Fill in last three rows and columns with constraints */
    for ( alt=1.0,j=0 ; j<=N ; j++ ) {
        c[j][N+1] = alt;
        c[j][N+2] = 1.0;
        c[j][N+3] = ((double) j*j);
        alt = -alt;
    }
    for ( alt=1.0,j=0 ; j<=N ; j++ ) {
        c[N+1][j] = alt;
        c[N+2][j] = 1.0;
        c[N+3][j] = ((double) j*j);
        alt = -alt;
    }
    b[N+2] = xmax;

/*  STEP 4. - Solve Equations */
    row = (int *) malloc( (N+M+1)*sizeof(int) );
    ludcmp( (N+M+1), c, row);
    linsol( N+M+1, c, a, b, row);

/*  STEP 5. - Create individual files */
    ptr = strstr( filename,"cell");
    k = 0;
    while ( *ptr != '_' ) {
        name[k++] = *ptr;
        ptr++;
    }
    name[k++] = '_';
    name[k++] = 'l';
    name[k++] = 'a';
    name[k++] = 'g';
    name[k++] = 'r';
    name[k++] = 'a';
    name[k++] = '.';
    name[k++] = 'R';
```

```
        name[k++] = 'E';
        name[k++] = 'S';
        name[k] = '\0';

/*  STEP 6. - Find value of function and derivative at given value */
        len = ((double) nobs);
        pi = 4.0*atan(1.0);
        fp = fopen(name,"w");
        for ( k=0 ; k<=nobs ; k++ )  {
            theta = 2.0*acos(sqrt(((double) k)/len));
            tmp = 0.0;
            for ( j=0 ; j<=N ; j++ ) {
                tmp += a[j]*cos(theta*((double) j));
            }
            diff = 0.0;
            for ( j=1 ; j<=N ; j++ ) {
                if ( fabs(theta) < 5.e-6 ) {
                    diff += a[j]*((double) j*j);
                } else if ( fabs(theta-pi) < 5.e-6 ) {
                    diff += a[j]*((double) j*j)*pow(-1.0, j+1);
                } else {
                    angle = theta*((double) j);
                    diff += a[j]*((double) j)*sin(angle)/sin(theta);
                }
            }
            diff *= 2.0/(len*EL*EU);
            fprintf(fp,"%d\t %lf\t %lf\n", k, tmp/xmax, diff/xmax);
        }
        fclose(fp);

/*  Clean up memory */
        free(row);
        free(a);
        free(b);
        for ( k=0 ; k<(N*M+1) ; k++ ) free(c[k]);
        free(c);
        free(x);
        free(t);
    }
    return(0);
}

/*****************************************************
                    LU Decomposition
*****************************************************/
int ludcmp( int n, double **a, int *row)
{
    double amax, tmp, sum, small=5.e-9, *ptr;
    int i, j, k, imax, kval;

/*  STEP 1. - Initialise row ordering */
    for ( j=0 ; j<n ; j++ ) row[j] = j;

/*  STEP 2. - Identify pivotal row */
    for ( j=0 ; j<n-1 ; j++ ) {
        imax = j;
        amax = fabs(a[j][j]);
        for ( i=j+1 ; i<n ; i++ ) {
            if ( (tmp=fabs(a[i][j])) > amax ) {
                amax = tmp;
                imax = i;
```

```
            }
        }
        if ( fabs(amax) < small ) return -1;

/*  STEP 3. - Interchange rows if necessary */
        if ( imax != j ) {
            kval = row[j];
            row[j] = row[imax];
            row[imax] = kval;
            ptr = a[j];
            a[j] = a[imax];
            a[imax] = ptr;
        }

/*  STEP 4. - Eliminate entries in column below (j,j)th entry */
        for ( i=j+1 ; i<n ; i++ ) {
            if ( a[i][j] != 0.0 ) {
                a[i][j] = a[i][j]/a[j][j];
                for ( k=j+1 ; k<n ; k++ ) a[i][k] -= a[i][j]*a[j][k];
            }
        }
    }
    if ( fabs(a[n-1][n-1]) < small  ) return -1;
    return 1;
}

/*******************************************************
    The solution function - apply in sequence with ludcmp
*******************************************************/
void linsol( int n, double **a, double *soln, double *b, int *row)
{
    double sum;
    int i, j, item;

/*  STEP 1. - Rearrange order of equations */
    for ( i=0 ; i<n ; i++ ) soln[i] = b[row[i]];

/*  STEP 2. - Forward substitution phase */
    for ( i=1 ; i<n ; i++ ) {
        for ( sum=0.0,j=0 ; j<i ; j++ ) sum += a[i][j]*soln[j];
        soln[i] -= sum;
    }

/*  STEP 3. - Backward substitution phase */
    soln[n-1] /= a[n-1][n-1];
    for ( i=n-2 ; i>=0 ; i-- ) {
        for ( sum=soln[i],j=i+1 ; j<n ; j++ ) sum -= a[i][j]*soln[j];
        soln[i] = sum/a[i][i];
    }
    return;
}
```

A.4 Building the typical neuron

The program developed to simulate neuronal morphology uses similar structures to the BuildNeuron.c program, however the morphological data has been extracted in a precursor program based on the BuildNeuron.c program. In summary MyeTypicalNeuron.c gathers the data required for each density estimate and then enters the recursive process, generating each branch diameter and length until completing the dendritic tree. Once the simulation has terminated, the morphological properties of the simulated cells are extracted in the final section of MyeTypicalNeuron.c.

A.4.1 MyeTypicalNeuron.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/*********************************************************************
           Program will construct the "Typical Neuron".

       Stage 1: Extracts neuron morphology data to construct the
                density estimates.
       Stage 2: Generates the typical neuron from the density
                functions.
       Stage 3: Extract morphological data from simulated cells.
*********************************************************************/

typedef struct unit_t
{
/*  Physical properties of unit */
    double len;                 /* Length of unit */
    double diam;                /* Diameter of unit */
    int    child;               /* Number of children */
    int    root;                /* Indicates if a root unit */
    int    term;                /* Indicates if terminal unit */
} unit;

/* Input files */
#define     INPUT1      "MyeBranch.dat"     /* Data file */
#define     INPUT2       "MyeCount.dat"     /* Count data */
#define     INPUT3    "MyeContDiam.dat"     /* P and C1 diameters */
#define     INPUT4  "MyeParentDiam.dat"     /* P, C1 and C2 diameters */
#define     INPUT5    "MyeRootDiam.dat"     /* Stem lengths & diameters */

/* Numerical Parameters */
#define      NCELL              31          /* No. of cells */
#define      NSEED               2          /* Random No. Seed */
#define       NSIM            1000          /* No. of simulations */
#define      PRINT               1          /* Print out data */

/* Global Functions */
double ran(unsigned long int *, unsigned long int *, unsigned long int *);
void heapsort( int, double * );
```

```
double scott_bandwidth( int, double, double * );
void scott_correlated( int, double *, double *, double *, double * );
double cdf1D( int, double * );
double pdf1D( int, double, double * );
double cdf2D( int, double, double *, double * );
double cdf3D( int, double, double, double *, double *, double * );
double phi( double );
void buildcell( double, branch *  );
void clean_dendrite( branch * );
void destroy_dendrite( branch * );
void Destroy_Sampled_Neuron(neuron **);
int count_branches( branch *, branch *);
int count_terminal_branches( branch *, branch *);
int branch_points( branch *, branch *);
void branch_length( FILE *, branch *);
void branchlen( FILE *, FILE *, branch *);
int count_root_branches( branch *, branch *);
int count_midsection_branches( branch *, branch *);
void branch_data( branch * );
void outputproperties( FILE *, branch * );
void output_branch( FILE *, branch * );
int count_unbranched( branch *, branch *);

/* Global Parameters */
unsigned long int ix, iy, iz;
int n, num, num1, num2, nt, nc, nb;
double sigma,pi,*ccdiam,*c1diam,*c2diam,*termd,*contd,*brand,*pcont,*all_len,*all_diam;

void main(void)
{
    extern unsigned long int ix, iy, iz;
    extern double sigma, *ccdiam, *c1diam, *c2diam, pi, *termd, *contd, *brand,
                  *all_len, *all_diam;
    extern int n, num, num1, num2, nlb, nlc;
    int c, d, j, k, nd, nr, **ndend, *p, nod, nm, start=1, np, nu;
    double *pdend, *rootd, diam, len, tmp;
    unit **mye;
    neuron **cell;
    FILE *fp, *fp1;

/*  STEP 1. - Initialisation */
    pi = 4.0*atan(1.0);
    srand( ((unsigned int) NSEED) );
    ix = rand( );
    iy = rand( );
    iz = rand( );

/*********************************************************************
        STAGE 1. - EXTRACT DATA
*********************************************************************/

/*  STEP 2. - Open myelinated data file */
    if ( (fp=fopen(INPUT1,"r")) != NULL ) {
        n = 0;

/*  STEP 2A. - Scan file to establish size and quantity of data */
        while ( fscanf(fp,"%lf %lf %d %d %d",&diam,&len,&p,&p,&p)!=EOF ) n++;

/*  STEP 2B. - Allocate memory to hold myelinated data values */
        mye = (unit **) malloc( n*sizeof(unit *) );
        for ( k=0 ; k<n ; k++ ) mye[k] = (unit *) malloc( sizeof(unit) );
```

```
        rewind(fp);

/*  STEP 2C. - Read data into vector */
        for ( k=0 ; k<n ; k++ ) {
            fscanf(fp, "%lf %lf %d %d %d", &(mye[k]->len), &(mye[k]->diam),
                       &(mye[k]->child),&(mye[k]->root),&(mye[k]->term) );
        }
        fclose(fp);
    } else {
        printf("\nCannot find input file!!");
        return;
    }
    all_len = (double *) malloc( n*sizeof(double) );
    all_diam = (double *) malloc( n*sizeof(double) );
    for ( k=0 ; k<n ; k++ ) {
        all_len[k] = mye[k]->len;
        all_diam[k] = mye[k]->diam;
    }
    fp=fopen("MyeSection.dat","w");
    for ( k=0 ; k<n ; k++ ) fprintf(fp,"%6.2lf\t%6.2lf\n", all_len[k], all_diam[k]);
    fclose(fp);

/*  STEP 3. - Open myelinated count data file */
    if ( (fp=fopen(INPUT2,"r")) != NULL ) {
        num = 0;

/*  STEP 3A. - Scan file to establish size and quantity of data */
        while ( fscanf(fp,"%d %d %d %d %d %d %d",&p,&p,&p,&p,&p,&p,&p)!=EOF) num++;

/*  STEP 3B. - Allocate memory to hold count data */
        ndend = (int **) malloc( 7*sizeof(int *) );
        for( k=0 ; k<7 ; k++ ) ndend[k]=(int *) malloc(num*sizeof(int));
        rewind(fp);

/*  STEP 3C. - Read data into vector */
        for ( k=0 ; k<num ; k++ ) {
            fscanf(fp, "%d %d %d %d %d %d %d",&ndend[0][k],&ndend[1][k],&ndend[2][k],
                              &ndend[3][k], &ndend[4][k], &ndend[5][k], &ndend[6][k]);
        }
        fclose(fp);
    } else {
        printf("\nCannot find input file!!");
        return;
    }

/*  STEP 4. - Open parent-ichild data file */
    if ( (fp=fopen(INPUT3,"r")) != NULL ) {
        nc = 0;

/*  STEP 4A. - Scan file to establish size and quantity of data */
        while ( fscanf(fp,"%lf %lf", &tmp, &tmp)!=EOF ) nc++;

/*  STEP 4B. - Allocate memory to hold parent-ichild data */
        contd = (double *) malloc( nc*sizeof(double) );
        ccdiam = (double *) malloc( nc*sizeof(double) );
        rewind(fp);

/*  STEP 4C. - Read data into vector */
        for(k=0 ; k<nc ; k++) fscanf(fp,"%lf %lf", &contd[k], &ccdiam[k]);
        fclose(fp);
    } else {
```

```
        printf("\nCannot find input file!!");
        return;
    }

/*  STEP 5. - Open parent-child1-child2 data file */
    if ( (fp=fopen(INPUT4,"r")) != NULL ) {
        nb = 0;

/*  STEP 5A. - Scan file to establish size and quantity of data */
        while ( fscanf(fp,"%lf %lf %lf", &tmp, &tmp, &tmp)!=EOF ) nb++;

/*  STEP 5B. - Allocate memory to hold p-c1-c2 data */
        brand = (double *) malloc( nb*sizeof(double) );
        c1diam = (double *) malloc( nb*sizeof(double) );
        c2diam = (double *) malloc( nb*sizeof(double) );
        rewind(fp);

/*  STEP 5C. - Read data into vector */
        for (k=0 ; k<nb ; k++) fscanf(fp,"%lf %lf %lf",&brand[k],&c1diam[k],&c2diam[k]);
        fclose(fp);
    } else {
        printf("\nCannot find input file!!");
        return;
    }

/*  STEP 6. - Get root length and diameter data */
    if ( (fp=fopen(INPUT5,"r")) != NULL ) {
        nr = 0;

/*  STEP 6A. - Allocate memory to hold length and diameter data */
        while ( fscanf(fp,"%lf", &rootd)!=EOF ) nr++;

/*  STEP 6B. - Allocate memory to hold p-c1-c2 data */
        rootd = (double *) malloc( nr*sizeof(double) );
        rewind(fp);

/*  STEP 6C. - Read data into vector */
        for ( k=0 ; k<nr ; k++ ) fscanf(fp, "%lf", &rootd[k]);
        fclose(fp);
    } else {
        printf("\nCannot find input file!!");
        return;
    }

/*  STEP 7. - Get remaining sections length and diameter data */
    nt = 0;
    for ( k=0 ; k<n ; k++ ) if ( mye[k]->child == 0 ) nt++;

/*  STEP 7A. - Allocate memory to hold length and diameter data */
    termd = (double *) malloc( nt*sizeof(double) );

/*  STEP 7B. - Read data into vector */
    j=0;
    for ( k=0 ; k<n ; k++ ) {
        if ( mye[k]->child == 0 ) {
            termd[j] = mye[k]->diam;
            j++;
        }
    }
    fp=fopen("MyeTerm.dat","w");
    for ( k=0 ; k<nt ; k++ ) fprintf(fp,"%8.2lf\n", termd[k]);
```

```
    fclose(fp);
    fp=fopen("MyeCont.dat","w");
    for ( k=0 ; k<nc ; k++ ) fprintf(fp,"%6.2lf\n", contd[k]);
    fclose(fp);
    fp=fopen("MyeBran.dat","w");
    for ( k=0 ; k<nb ; k++ ) fprintf(fp,"%6.2lf\n", brand[k]);
    fclose(fp);

/*  STEP 8. - Calculate probability of dendrites per cell */
    nd = 0;
    for ( k=0 ; k<num ; k++ ) if ( ndend[0][k] > nd ) nd = ndend[0][k];
    p = (int *) malloc ( (nd+1)*sizeof(int) );
    pdend = (double *) malloc ( (nd+1)*sizeof(double) );
    for ( k=0 ; k<=nd ; k++ ) {
        pdend[k] = 0.0;
        p[k] = 0;
    }
    for ( j=1 ; j<=nd ; j++ ) {
        for ( k=0 ; k<num ; k++ ) if ( ndend[0][k] == j ) p[j]++;
    }
    for ( k=1 ; k<=nd ; k++ ) p[0] += p[k];
    for ( k=0 ; k<=nd ; k++ ) pdend[k] = ((double) p[k]);
    for ( k=1 ; k<=nd ; k++ ) pdend[k] = pdend[k]/pdend[0];
    pdend[0] = 0.0;
    for ( k=1 ; k<nd ; k++ ) pdend[k] += pdend[k-1];
    pdend[nd] = 1.0;

/*************************************************************************
        STAGE 2. - BUILD CELLS
*************************************************************************/
    pcont = (double *) malloc( 4*sizeof(double) );
    cell = (neuron **) malloc( NCELL*sizeof(neuron *) );
    for ( k=0 ; k<NCELL ; k++ ) cell[k] =(neuron *)malloc(sizeof(neuron));
    for ( c=0 ; c<NCELL ; c++ ) {

/*  STEP 10. - Calculate number of dendrites */
        tmp = ran( &ix, &iy, &iz);
        for ( k=0 ; k<nd ; k++ ) {
            if ( tmp >= pdend[k] && tmp <= pdend[k+1] ) nod = k+1;
        }
        cell[c]->ndend = nod;
        cell[c]->dendlist = (dendrite *) malloc((cell[c]->ndend)*sizeof(dendrite));
        printf("\nCell has %d dendrites", cell[c]->ndend);

/*  STEP 11. - Run through dendrites */
        for ( d=0 ; d<nod ; d++ ) {

/*  STEP 12. - Get root diameter */
            diam = cdf1D( nr, rootd );
            printf("\nRoot section has diameter %lf", diam);

/*  STEP 13. - Get root length conditioned by root diameter */
            len = cdf2D( n, diam, all_diam, all_len);
            printf("\nRoot section has length %lf", len);
            cell[c]->dendlist[d].root=(branch *)malloc(sizeof(branch));
            cell[c]->dendlist[d].root->d=(double *)malloc(1*sizeof(double));
            cell[c]->dendlist[d].root->plen=(double*)malloc(1*sizeof(double));
            cell[c]->dendlist[d].root->nobs = 1;
            cell[c]->dendlist[d].root->d[0] = diam;
            cell[c]->dendlist[d].root->plen[0] = len;
            cell[c]->dendlist[d].root->parent = NULL;
```

```
            cell[c]->dendlist[d].root->child = NULL;
            cell[c]->dendlist[d].root->peer = NULL;

/*  STEP 14. - Generate coordinates */
            for ( k=0 ; k<3 ; k++ ) {
                cell[c]->dendlist[d].root->f[k] = 0.0;
                cell[c]->dendlist[d].root->l[k] = ran( &ix, &iy, &iz);
            }

/*  STEP 15. - Enter recursive routine to construct dendrite */
            buildcell( diam, cell[c]->dendlist[d].root );
            printf("\nCompleted cell[%d]->dendlist[%d]\n", c,d);
        }
    }

/*  STEP 16. - Clean Dendrite: turn sections into branches */
    for ( c=0 ; c<NCELL ; c++ ) {
        for (d=0;d<(cell[c]->ndend);d++) clean_dendrite(cell[c]->dendlist[d].root);
    }
    printf("\nCleaned dendrite");

/*  STEP 17. - Get total branch lengths */
    fp=fopen("MSimBranchLen.dat","w");
    fp1=fopen("MSimUnBranLen.dat","w");
    for ( c=0 ; c<NCELL ; c++ ) {
        for (d=0;d<(cell[c]->ndend);d++) branchlen(fp,fp1,cell[c]->dendlist[d].root);
    }
    fclose(fp);
    fclose(fp1);

/*  STEP 18. - Get dendritic lengths  ******************************/
    fp=fopen("MSimDendLen.dat","w");
    for ( c=0 ; c<NCELL ; c++ ) {
        for (d=0; d<(cell[c]->ndend) ; d++) branch_length(fp,cell[c]->dendlist[d].root);
    }
    fclose(fp);

/*  STEP 9. - Output branch properties to construct the dendogram */
    fp=fopen("MSimBranchProp.dat","w");
    for ( c=0 ; c<NCELL ; c++ ) {
        fprintf(fp,"Cell %d\n",c);
        for ( d=0 ; d<(cell[c]->ndend) ; d++ ) {
            output_branch( fp, cell[c]->dendlist[d].root );
            fprintf(fp,"\n");
        }
    }
    fclose(fp);

/*  STEP 9. - Output branch properties to construct the dendogram */
    fp=fopen("MSimSection.dat","w");
    for ( c=0 ; c<NCELL ; c++ ) {
        for ( d=0 ; d<(cell[c]->ndend) ; d++ ) {
            outputproperties( fp, cell[c]->dendlist[d].root );
        }
    }
    fclose(fp);

/*************************************************************************
        STAGE 3. - EXTRACT DATA FROM SIMULATED CELLS
*************************************************************************/
```

```
/*  STEP 19. - Count branches */
    for ( c=0 ; c<NCELL ; c++ ) {
        for ( nb=k=0 ; k<cell[c]->ndend ; k++ )
        nb += count_branches( cell[c]->dendlist[k].root,cell[c]->dendlist[k].root);
        printf("\n Found %d branches", nb);

/*  STEP 20. - Count root branches */
        for ( nr=k=0 ; k<cell[c]->ndend ; k++ )
        nr += count_root_branches (cell[c]->dendlist[k].root,cell[c]->dendlist[k].root);
        printf("\n Found %d root branches", nr);

/*  STEP 21. - Count mid section branches */
        for ( nm=k=0 ; k<cell[c]->ndend ; k++ )
        nm += count_midsection_branches (
                                cell[c]->dendlist[k].root, cell[c]->dendlist[k].root);
        printf("\n Found %d mid section branches", nm);

/*  STEP 22. - Count terminal branches */
        for ( nt=k=0 ; k<cell[c]->ndend ; k++ )
        nt += count_terminal_branches
                (cell[c]->dendlist[k].root, cell[c]->dendlist[k].root);
        printf("\n Found %d terminal branches\n", nt);

/*  STEP 22. - Count unbranched branches */
        for ( nu=k=0 ; k<cell[c]->ndend ; k++ )
        nu += count_unbranched
                (cell[c]->dendlist[k].root, cell[c]->dendlist[k].root);
        printf("\n Found %d unbranched branches\n", nu);

/*  STEP 22. - Count branch points */
        for ( np=k=0 ; k<cell[c]->ndend ; k++ )
        np += branch_points( cell[c]->dendlist[k].root, cell[c]->dendlist[k].root);
        printf("\n Found %d branch points\n", np);

/*  STEP 23. - Output count data */
        if ( start ) {
            fp = fopen("MSimCount.dat","w");
            fprintf(fp, "%3d\t %3d\t %3d\t %3d\t %3d\t %3d\t %3d\n",
                                cell[c]->ndend, nb, np, nr, nm, nt, nu);
            start = 0;
        } else {
            fp = fopen("MSimCount.dat","a");
            fprintf(fp, "%3d\t %3d\t %3d\t %3d\t %3d\t %3d\t %3d\n",
                                cell[c]->ndend, nb, np, nr, nm, nt, nu);
        }
        fclose(fp);

/*  STEP 25. - Output lengths and diameter */
        for ( k=0 ; k<cell[c]->ndend ; k++ ) branch_data( cell[c]->dendlist[k].root );
    }

/*  STEP 26. - Tidy up */
    Destroy_Sampled_Neuron( cell );
    for ( k=0 ; k<n ; k++ ) free(mye[k]);
    free(mye);
    for ( k=0 ; k<7 ; k++ ) free(ndend[k]);
    free(ndend);
    free(pdend);
    free(rootd);
    free(c1diam);
    free(c2diam);
```

```
    free(ccdiam);
    free(contd);
    free(brand);
    free(termd);
    free(pcont);
    free(p);
    free(all_len);
    free(all_diam);
    return;
}

/*************************************************************************
        FUNCTION TO BUILD THE DENDRITE
*************************************************************************/
void buildcell( double diam, branch *b  )
{
    extern double sigma, *ccdiam, *c1diam, *c2diam, *brdiam, *pcdiam,
           pi, *termd, *contd, *brand, *pcont, *all_len, *all_diam;
    extern int n, num, numi, num2, nt, nc, nb;
    int k, next;
    double l, d, pt, pc, pb, t, c, br, tmp, fract, fracc, fracb, d1, d2;
    branch *bnow, *btmp;

    fract = ((double) nt)/((double) nt+nc+nb);
    fracc = ((double) nc)/((double) nt+nc+nb);
    fracb = ((double) nb)/((double) nt+nc+nb);

    t = fract*pdf1D( nt, diam, termd );
    c = fracc*pdf1D( nc, diam, contd );
    br = fracb*pdf1D( nb, diam, brand );

    pt = (t/(t+c+br));
    pc = (c/(t+c+br));
    pb = (br/(t+c+br));
    pcont[0] = 0.0;
    pcont[1] = pt;
    pcont[2] = pt+pc;
    pcont[3] = pt+pc+pb;
    tmp = ran( &ix, &iy, &iz);
    for ( k=0 ; k<3 ; k++ ) {
        if ( tmp >= pcont[k] && tmp <= pcont[k+1] ) next = k;
    }
    printf("\nSection has %d child(ren)", next);

    if ( next == 1 ) {                              // Continuos
        bnow = (branch *) malloc( sizeof(branch) );
        bnow->d = (double *) malloc( 1*sizeof(double) );
        bnow->plen = (double *) malloc( 1*sizeof(double) );
        d = cdf2D( nc, diam, contd, ccdiam );
        printf("\nSection has diam %lf", d);
        l = cdf2D( n, d, all_diam, all_len );
        printf("\nSection has len %lf", l);
        b->child = bnow;
        bnow->parent = b;
        bnow->child = NULL;
        bnow->peer = NULL;
        bnow->plen[0] = l;
        bnow->d[0] = d;
        bnow->nobs = 1;
        for ( k=0 ; k<3 ; k++ ) bnow->f[k] = b->l[k];
        for ( k=0 ; k<3 ; k++ ) bnow->l[k] = ran( &ix, &iy, &iz);
```

```
        buildcell( d, bnow );
    } else if ( next == 2 ) {                          // Branches
        bnow = (branch *) malloc( sizeof(branch) );
        bnow->d = (double *) malloc( 1*sizeof(double) );
        bnow->plen = (double *) malloc( 1*sizeof(double) );
        d1 = cdf2D( nb, diam, brand, c1diam);
        l = cdf2D( n, d1, all_diam, all_len );
        b->child = bnow;
        bnow->parent = b;
        bnow->child = NULL;
        bnow->d[0] = d1;
        bnow->plen[0] = l;
        bnow->nobs = 1;
        for ( k=0 ; k<3 ; k++ ) bnow->f[k] = b->l[k];
        for ( k=0 ; k<3 ; k++ ) bnow->l[k] = ran( &ix, &iy, &iz);
        btmp = (branch *) malloc( sizeof(branch) );
        btmp->d = (double *) malloc( 1*sizeof(double) );
        btmp->plen = (double *) malloc( 1*sizeof(double) );
        d2 = cdf3D( nb, diam, d1, brand, c1diam, c2diam );
        l = cdf2D( n, d2, all_diam, all_len );
        btmp->d[0] = d2;
        btmp->plen[0] = l;
        btmp->parent = b;
        btmp->peer = NULL;
        btmp->child = NULL;
        bnow->peer = btmp;
        btmp->nobs = 1;
        for ( k=0 ; k<3 ; k++ ) btmp->f[k] = b->l[k];
        for ( k=0 ; k<3 ; k++ ) btmp->l[k] = ran( &ix, &iy, &iz);
        buildcell( d1, bnow );
        buildcell( d2, btmp );
    } else {                                           // Terminates
        b->child = NULL;
        return;
    }
    return;
}

/*************************************************************************
        CALCULATES CDF AND RETURNS A DEVIATE - ONE-DIMENSION
*************************************************************************/
double cdf1D( int n, double *val )
{
    extern unsigned long int ix, iy, iz;
    int j, k;
    double cdf, min, max, mid, sum, h;

    min = max = val[0];
    for ( k=1 ; k<n ; k++ ) {
        if ( val[k] < min ) min = val[k];
        if ( val[k] > max ) max = val[k];
    }
    h = scott_bandwidth( n, 1.0, val );

    cdf = 0.5+0.5*ran( &ix, &iy, &iz);
    min -= 10.0*h;
    max += 10.0*h;
    do {
        mid = 0.5*(min+max);
        for ( sum=0.0,j=0 ; j<n ; j++ ) sum += phi( (mid-val[j])/h );
        for ( j=0 ; j<n ; j++ ) sum += phi( (mid+val[j])/h );
```

```
        sum /= ((double) 2*n);
        if ( sum > cdf ) {
            max = mid;
        } else {
            min = mid;
        }
    } while ( max-min > 5.e-7 );
    mid = 0.5*(min+max);

    return(mid);
}

/*************************************************************************
        CALCULATES PDF AND RETURNS A DEVIATE - ONE-DIMENSION
*************************************************************************/
double pdf1D( int n, double diam, double *val )
{
    int k;
    double ker, tmp, fac, h;

    h = scott_bandwidth( n, 1.0, val );
    fac = 1.0/(h*sqrt(2.0*pi)*((double) n));
    ker = 0.0;
    for ( k=0 ; k<n ; k++ ) {
        tmp = 0.5*pow((diam-val[k])/h, 2);
        if ( tmp < 20.0 ) ker += exp(-tmp);
    }
    ker *= fac;

    return(ker);
}

/*************************************************************************
        CALCULATES CONDITIONAL DENSITY AND RETURNS DEVIATE
*************************************************************************/
double cdf2D( int n, double diam, double *x, double *y )
{
    extern unsigned long int ix, iy, iz;
    int j, k;
    double *w, cdf, min, max, mid, sum, hx, hy, tmp;

    w = (double *) malloc( n*sizeof(double) );
    for ( k=0 ; k<n ; k++ ) w[k] = 0.0;

    scott_correlated( n, &hx, &hy, x, y );

    for ( k=0 ; k<n ; k++ ) {
        tmp = (diam-x[k])/hx;
        tmp *= tmp;
        if ( tmp < 50.0 ) w[k] = exp(-0.5*tmp);
        tmp = (diam+x[k])/hx;
        tmp *= tmp;
        if ( tmp < 50.0 ) w[k] += exp(-0.5*tmp);
    }
    for ( sum=0.0, k=0 ; k<n ; k++ ) sum += w[k];
    for ( k=0 ; k<n ; k++ ) w[k] /= sum;

    min = max = y[0];
    for ( k=1 ; k<n ; k++ ) {
        if ( y[k] < min ) min = y[k];
        if ( y[k] > max ) max = y[k];
```

```
    }

    cdf = 1.0+ran( &ix, &iy, &iz);
    min -= 10.0*hy;
    max += 10.0*hy;
    do {
        mid = 0.5*(min+max);
        for( sum=0.0,j=0 ; j<n ; j++ ) sum += w[j]*phi((mid-y[j])/hy);
        for( j=0 ; j<n ; j++ ) sum += w[j]*phi( (mid+y[j])/hy );
        if ( sum > cdf ) {
            max = mid;
        } else {
            min = mid;
        }
    } while ( max-min > 5.e-7 );
    mid = 0.5*(min+max);

    free(w);
    return(mid);
}

/*************************************************************************
        CALCULATES 3D CONDITIONAL DENSITY AND RETURNS DEVIATE
*************************************************************************/
double cdf3D(int n,double diam,double di,double *p,double *c1,double *c2)
{
    extern unsigned long int ix, iy, iz;
    int j, k;
    double *w, cdf, c2min, c2max, c2mid, sum, hx, hy, h, tmp, d2;

    w = (double *) malloc( n*sizeof(double) );
    for ( k=0 ; k<n ; k++ ) w[k] = 0.0;

    scott_correlated( n, &hx, &hy, p, c1 );
    for ( k=0 ; k<n ; k++ ) {
        tmp = (diam-p[k])/hx;
        tmp *= tmp;
        if ( tmp < 50.0 ) w[k] = exp(-0.5*tmp);
        tmp = (diam+p[k])/hx;
        tmp *= tmp;
        if ( tmp < 50.0 ) w[k] += exp(-0.5*tmp);
    }
    for ( sum=0.0, k=0 ; k<n ; k++ ) sum += w[k];
    for ( k=0 ; k<n ; k++ ) w[k] /= sum;

    for ( k=0 ; k<n ; k++ ) {
        sum = 0.0;
        tmp = (di-c1[k])/hy;
        tmp *= tmp;
        if ( tmp < 50.0 ) sum = exp(-0.5*tmp);
        tmp = (di+c1[k])/hy;
        tmp *= tmp;
        if ( tmp < 50.0 ) sum += exp(-0.5*tmp);
        w[k] *= sum;
    }
    for ( sum=0.0, k=0 ; k<n ; k++ ) sum += w[k];
    for ( k=0 ; k<n ; k++ ) w[k] /= sum;

    h = scott_bandwidth( n, 3.0, c2 );
    c2min = c2max = c2[0];
    for ( k=1 ; k<n ; k++ ) {
```

```
        if ( c2[k] < c2min ) c2min = c2[k];
        if ( c2[k] > c2max ) c2max = c2[k];
    }

    cdf = 1.0+ran( &ix, &iy, &iz);
    c2min -= 10.0*h;
    c2max += 10.0*h;
    do {
        c2mid = 0.5*(c2min+c2max);
        for( sum=0.0,j=0 ; j<n ; j++ ) sum += w[j]*phi((c2mid-c2[j])/h);
        for( j=0 ; j<n ; j++ ) sum += w[j]*phi( (c2mid+c2[j])/h );
        if ( sum > cdf ) {
            c2max = c2mid;
        } else {
            c2min = c2mid;
        }
    } while ( c2max-c2min > 5.e-7 );
    d2 = 0.5*(c2min+c2max);

    free(w);
    return(d2);
}


/*************************************************************************
        Function to find length of branch
*************************************************************************/
void branchlen( FILE *fp, FILE *fp1, branch *bnow )
{
    int k;
    double length;

    if ( bnow->child ) branchlen( fp, fp1, bnow->child);
    if ( bnow->peer ) branchlen( fp, fp1, bnow->peer);

    for ( length=0.0,k=0 ; k<(bnow->nobs) ; k++ ) {
        length += bnow->plen[k];
    }
    bnow->len = length;
    fprintf(fp,"%10.6lf\t%6.2lf\n", bnow->len, bnow->d[0]);

    return;
}

/*************************************************************
        Function to count number of root branches and print
        the length of the branches out to file.
*************************************************************/
int count_root_branches( branch *bstart, branch *bnow)
{
    static int n;

    if ( bstart == bnow ) n = 0;
    if ( bnow ) {
        if ( bnow->child ) count_root_branches(bstart, bnow->child);
        if ( bnow->peer ) count_root_branches(bstart, bnow->peer);
        if ( !bnow->parent ) n++;
    }
    return n;
}
```

```
/*************************************************************
        Function to count number of root branches and print
        the length of the branches out to file.
*************************************************************/
int count_midsection_branches( branch *bstart, branch *bnow)
{
    static int n;

    if ( bstart == bnow ) n = 0;
    if ( bnow ) {
        if (bnow->child)count_midsection_branches(bstart,bnow->child);
        if (bnow->peer) count_midsection_branches(bstart, bnow->peer);
        if (bnow->parent && bnow->child) n++;
    }
    return n;
}

/*************************************************************
        Function to count number of root branches and print
        the length of the branches out to file.
*************************************************************/
int count_unbranched( branch *bstart, branch *bnow)
{
    static int n;

    if ( bstart == bnow ) n = 0;
    if ( bnow ) {
        if ( bnow->child ) count_unbranched(bstart, bnow->child);
        if ( bnow->peer ) count_unbranched(bstart, bnow->peer);
        if ( !bnow->parent && !bnow->child ) n++;
    }
    return n;
}

/*************************************************************
        Function to count number of root branches and print
        the length of the branches out to file.
*************************************************************/
int branch_points( branch *bstart, branch *bnow)
{
    static int n;

    n = 0;
    if ( bnow ) {
        if ( bnow->child ) branch_points(bstart, bnow->child);
        if ( bnow->peer ) branch_points(bstart, bnow->peer);
        if ( bnow->child && bnow->child->peer ) n++;
    }
    return n;
}


/*************************************************************
         Function to count number of child branches
*************************************************************/
void branch_data( branch *b )
{
    static int k, initial=1, start=1, first=1, first1=1,
               flag, n, num, sect, root, child;
    double dold, dnew, len, xold, xnew;
    branch *bran;
```

```
    FILE *fp, *fp1, *fp2, *fp3;

    if ( b->child ) branch_data( b->child );
    if ( b->peer ) branch_data( b->peer );

/*  Initialisation phase */
    n = 0;
    if ( start ) {
        fp = fopen("MSimBranch.dat","w");
        start = 0;
        num = 1;
    } else {
        fp = fopen("MSimBranch.dat","a");
        num++;
    }

/*  Count number of child branches */
    bran = b->child;
    while ( bran != NULL ) {
        n++;
        bran = bran->peer;
    }

/*  Determine type of branch: root(0), midsection(1), or terminal(2) */
    if ( !b->parent ) {
        root = 1;
    } else {
        root = 0;
    }
    if ( !b->parent ) {
        sect = 0;
    } else if ( b->parent && b->child ) {
        sect = 1;
    } else if ( !b->child ) {
        sect = 2;
    }

/*  Print out parent and child diameters */
    if ( first1 ) {
        fp2=fopen("MSimParentDiam.dat","w");
        first1 = 0;
    } else {
        fp2=fopen("MSimParentDiam.dat","a");
    }

/*  Print out root diameters */
    if ( first ) {
        fp1=fopen("MSimRootDiam.dat","w");
        first = 0;
    } else {
        fp1=fopen("MSimRootDiam.dat","a");
    }
    if ( !b->parent && b->child ) fprintf(fp1,"%4.2lf\n", b->d[0]);
    flag = 1;

/*  Print out parent & one child diameters */
    if ( initial ) {
        fp3=fopen("MSimContDiam.dat","w");
        initial = 0;
    } else {
        fp3=fopen("MSimContDiam.dat","a");
```

```
    }

/*  Decompose branches into lengths of uniform diameter */
    len = xold = b->plen[0];
    dold = b->d[0];
    for ( k=1 ; k<b->nobs ; k++ ) {
        xnew = b->plen[k];
        dnew = b->d[k];
        if ( dnew != dold ) {
            child = 1;
            fprintf(fp,"%6.2lf\t %6.2lf\t %3d\t %3d\t %3d\n",
                                        len, dold, child, sect, num);
//            if ( root && flag ) {
//                fprintf(fp1,"%6.2lf\t%6.2lf\n", len, dold);
//                flag = 0;
//            }
            fprintf(fp3,"%6.2lf\t%6.2lf\n", dold, dnew);
            len = xnew;
        } else {
//            if ( root && flag ) {
//                fprintf(fp1,"%6.2lf\t%6.2lf\n", len, dold);
//                flag = 0;
//            }
            len += xnew;
        }
        xold = xnew;
        dold = dnew;
    }
    fprintf(fp,"%6.2lf\t %6.2lf\t %3d\t %3d\t %3d\n",
                                        len, dold, n, sect, num);

    if ( b->child && b->child->peer ) {
        if (  b->child->d[0] > b->child->peer->d[0] ) {
            fprintf(fp2,"%6.2lf\t %6.2lf\t %6.2lf\n",
            b->d[b->nobs-1], b->child->d[0], b->child->peer->d[0]);
        } else {
            fprintf(fp2,"%6.2lf\t %6.2lf\t %6.2lf\n",
                b->d[b->nobs-1],b->child->peer->d[0],b->child->d[0]);
        }
    }
    fclose(fp);
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);

    return;
}
```

A.4.2 BiVarKernels.c

This program estimates the bivariate kernel density of two data sets, for example, parent and child section diameter. The procedure is described in Chapter 5. This program uses the correlated Scott bandwidth calculation described in Chapter 5.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/*********************************************************************
            Construct bivariate densities.
*********************************************************************/

/* Input files */
#define        INPUT        "MSimParentDiam.dat"     /* Input file */
#define       OUTPUT       "MSimChildrenKer.dat"     /* Output file */
#define      OUTPUT1    "MSimChildrenCoords.dat"     /* Output file */

/* Numerical Parameters */
#define          NSIM            10     /* No. of simulations */
#define           DIM             2     /* Dim's - no of samples */
#define           VAR             2     /* No of variables in file*/
#define         NSEED            20     /* Random No. Seed */

/* Global Functions */
void heapsort( int, double * );
double scott_bandwidth( int, double * );
void scott_correlated( int, double *, double *, double *, double * );

/* Global Parameters */
double *z, *d, anew, sigma, sigmax, sigmay;
int ndim;

void main(void)
{
    extern double *z, *d, anew, sigmax, sigmay;
    extern int ndim;
    int i, j, k, s, t, num;
    double *val1, *val2, *data, x, y, pi, tmp, tmp1, tmp2, hx, hy,
           first1, last1, first2, last2, dx, dy, ker, fac, v;
    FILE *fp, *fp1, *fp2;

/*  STEP 1. - Open myelinated data file */
    if ( (fp=fopen(INPUT,"r")) != NULL ) {
        num = 0;

/*  STEP 1a. - Scan file to establish size and quantity of data */
        while ( fscanf(fp,"%lf %lf", &tmp, &tmp)!=EOF ) num++;

/*  STEP 1b. - Allocate memory to hold myelinated data values */
        val1 = (double *) malloc( num*sizeof(double) );
        val2 = (double *) malloc( num*sizeof(double) );
        rewind(fp);

/*  STEP 1c. - Read data into vector */
        for ( k=0 ; k<num ; k++ ) {
```

```
            fscanf(fp, "%lf %lf", &val1[k], &val2[k]);
        }
        fclose(fp);
    } else {
        printf("\nCannot find input file!!");
        return;
    }
    printf("\n%d items in %s", num, INPUT);

/*  STEP 2. - Myelinated cells */
    data = (double *) malloc( num*sizeof(double) );
    for ( k=0 ; k<num ; k++ ) data[k] = 0.0;

    scott_correlated( num, &hx, &hy, val1, val2 );

/*  STEP 3. - Calculate bandwidth for lengths */
    for ( k=0 ; k<num ; k++ ) {
        data[k] = val1[k];
        heapsort( num, data );
    }

    first1 = data[0]-(2.0*sigmax);
    if ( first1 < 0.0 ) first1 = 0.0;
    last1 = data[num-1]+(2.0*sigmax);
    dx = (last1-first1)/100.0;

/*  STEP 4. - Calculate bandwidth for lengths */
    for ( k=0 ; k<num ; k++ ) {
        data[k] = val2[k];
        heapsort( num, data );
    }

    first2 = data[0]-(2.0*sigmay);
    if ( first2 < 0.0 ) first2 = 0.0;
    last2 = data[num-1]+(2.0*sigmay);
    dy = (last2-first2)/100.0;

/*  STEP 5. - Calculate density */
    fac = 9.0/(2000.0*hx*hy*((double) num));
    fp = fopen(OUTPUT,"w");
    fp1 = fopen(OUTPUT1,"w");
    for ( j=0 ; j<=100 ; j++ ) {
        x = first1+dx*((double) j);
        for ( k=0 ; k<=100 ; k++ ) {
            ker = 0.0;
            y = first2+dy*((double) k);
            for ( t=0 ; t<num ; t++ ) {
                tmp1 = fabs((x-val1[t])/hx);
                tmp2 = fabs((y-val2[t])/hy);
                if ( tmp1 <= sqrt(5.0) && tmp2 <= sqrt(5.0) ) {
                    ker += (5.0-tmp1*tmp1)*(5.0-tmp2*tmp2);
                }
            }
            ker *= fac;
            if ( j == k ) fprintf(fp1, "%12.6lf\t", y);
            fprintf(fp, "%12.6lf\t", ker);
        }
        fprintf(fp,"\n");
        fprintf(fp1, "%12.6lf\n", x);
    }
    fclose(fp);
```

```
    fclose(fp1);

/*  STEP 6. - Tidy up */
    free(val1);
    free(val2);
    free(data);

    return;
}
```

A.4.3 SimStats.c

To analyse the results of the simulation exercise, the following program SimStats.c was developed. It calculates the mean value and standard deviation of the density of each parameter at specified values. For example, each simulation will return an estimated density for dendritic length and by applying SimStats.c, the mean density and corresponding standard deviation can be calculated. As displayed in Chapter 5, this can be used to compare the real density with the simulated density when testing the results of the simulation.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

/*************************************************************************
        Program to calculate the mean and standard deviation of
        the estimated densities from the simulation program.
*************************************************************************/

/* Numerical Parameters */
#define       NAME       "MSIMUBDiaSimKer.dat"    /* Input file */
#define     OUTPUT     "MSIMUBDiamKernels.dat"    /* Output file */
#define       NSIM                         200    /* No. of simul's */
#define       NKER                         101    /* No. of div's */

void main(void)
{
    int j, k;
    double *mu, *sigma, **den, *max, *min, mean, sd, h, dx,
           pi, fac, ker, tmp, x, first, last;
    FILE *fp;

/*  STEP 1. - Determine file exists */
    if ( (fp=fopen(NAME,"r")) != NULL ) {
        den = (double **) malloc( NSIM*sizeof(double *) );
        for ( k=0 ; k<NSIM ; k++ ) den[k] = (double *) malloc( NKER*sizeof(double) );
        for ( k=0 ; k<NSIM ; k++ ) {
            for ( j=0 ; j<NKER ; j++ ) fscanf(fp,"%lf", &den[k][j]);
        }
        fclose(fp);
    } else {
        printf("\nCannot find input file!!");
        return;
    }

/*  STEP 2. - Calculate mean and standard deviation */
    mu = (double *) malloc( NKER*sizeof(double) );
    sigma = (double *) malloc( NKER*sizeof(double) );
    for ( k=0 ; k<NKER ; k++ ) mu[k] = sigma[k] = 0.0;

    for ( k=0 ; k<NKER ; k++ ) {
        for ( j=0 ; j<NSIM ; j++ ) {
            mu[k] += den[j][k];
            sigma[k] += pow(den[j][k], 2);
        }
        mu[k] /= ((double) NSIM);
```

```
        sigma[k] /= ((double) NSIM);
        sigma[k] = sqrt(sigma[k]-mu[k]*mu[k]);
    }

/*  STEP 3. - Calculate and draw max and min curves */
    max = (double *) malloc( NKER*sizeof(double) );
    min = (double *) malloc( NKER*sizeof(double) );
    for ( k=0 ; k<NKER ; k++ ) max[k] = min[k] = 0.0;

    for ( k=0 ; k<NKER ; k++ ) {
        max[k] = mu[k]+2.0*sigma[k];
        if ( max[k] < 0.0 ) max[k] = 0.0;
        min[k] = mu[k]-2.0*sigma[k];
        if ( min[k] < 0.0 ) min[k] = 0.0;
    }

    fp=fopen(OUTPUT,"w");
    for (k=0;k<NKER;k++) fprintf(fp,"%10.8lf\t%10.8lf\t%10.8lf\n",
                                 1000*min[k], 1000*mu[k], 1000*max[k]);
    fclose(fp);

/*  STEP 4. - Tidy up */
    for ( k=0 ; k<NSIM ; k++ ) free(den[k]);
    free(den);
    free(mu);
    free(sigma);
    free(min);
    free(max);
    return;
}
```

A.5 Hodgkin-Huxley rate functions

The auxiliary variables m, n and h in the Hodgkin-Huxley membrane model define the kinetic behaviour of the conductances through differential equations of the form

$$\frac{dy}{dt} = \alpha_y (1 - y) - \beta_y y$$

where $y = h, m, n$ and α and β are functions of V_M. The following code calculates the values of α and β for a specified voltage for each of the auxiliary variables h, m and n.

```
/*************************************************************
                    ALPHA for ACTIVATION OF SODIUM
*************************************************************/
double alfa_m( double volt )
{
    double tmp;
    static double fac;
    static int start=1;

    if ( start ) {
        fac = pow(3.0,0.1*CELSIUS-0.63);
        start = !start;
    }
    tmp = -0.1*(volt+35.0);
    if ( fabs(tmp)<0.001 ) {
        tmp = 1.0/(((tmp/24.0+1.0/6.0)*tmp+0.5)*tmp+1.0);
    } else {
        tmp = tmp/(exp(tmp)-1.0);
    }
    return tmp*fac;
}

/*************************************************************
                    BETA for ACTIVATION OF SODIUM
*************************************************************/
double beta_m( double volt )
{
    double tmp;
    static double fac;
    static int start=1;

    if ( start ) {
        fac = pow(3.0,0.1*CELSIUS-0.63);
        start = !start;
    }
    tmp = (volt+60.0)/18.0;
    return 4.0*fac*exp(-tmp);
}

/*************************************************************
                    ALPHA for INACTIVATION OF SODIUM
*************************************************************/
double alfa_h( double volt )
{
    double tmp;
```

```
    static double fac;
    static int start=1;

    if ( start ) {
        fac = pow(3.0,0.1*CELSIUS-0.63);
        start = !start;
    }
    tmp = 0.05*(volt+60.0);
    return 0.07*fac*exp(-tmp);
}

/*****************************************************************
                    BETA for INACTIVATION OF SODIUM
 *****************************************************************/
double beta_h( double volt )
{
    double tmp;
    static double fac;
    static int start=1;

    if ( start ) {
        fac = pow(3.0,0.1*CELSIUS-0.63);
        start = !start;
    }
    tmp = -0.1*(volt+30.0);
    return fac/(exp(tmp)+1.0);
}

/*****************************************************************
                    ALPHA for ACTIVATION OF POTASSIUM
 *****************************************************************/
double alfa_n( double volt )
{
    double tmp;
    static double fac;
    static int start=1;

    if ( start ) {
        fac = pow(3.0,0.1*CELSIUS-0.63);
        start = !start;
    }
    tmp = -0.1*(volt+50.0);
    if ( fabs(tmp)<0.001 ) {
        tmp = 0.1/(((tmp/24.0+1.0/6.0)*tmp+0.5)*tmp+1.0);
    } else {
        tmp = 0.1*tmp/(exp(tmp)-1.0);
    }
    return tmp*fac;
}

/*****************************************************************
                    BETA for ACTIVATION OF POTASSIUM
 *****************************************************************/
double beta_n( double volt )
{
    double tmp;
    static double fac;
    static int start=1;

    if ( start ) {
        fac = pow(3.0,0.1*CELSIUS-0.63);
```

```
        start = !start;
    }
    tmp = 0.0125*(volt+60.0);
    return 0.125*fac*exp(-tmp);
}
```

A.6 Utility programs

This section contains some standard functions that are common to many of the programs, for example a random number generator and a number sorting algorithm.

```
/*************************************************************
    Calculate the cumulative normal distribution phi(z)
*************************************************************/
double phi(double z)
{
    double t, x, y, hrt=0.7071067811865475;

    x = -z*hrt;
    if ( x >= 26.6 ) return 0.0;
    if ( x <= -6.5 ) return 1.0;
    t = 1.0-7.5/(fabs(x)+3.75);
    y = ((((((((((((-1.5802348811965169e-11*t
        -4.9497206900939292e-11)*t+1.8642495354462378e-10)*t
        +6.2979624691823961e-10)*t-1.3475134097349389e-9)*t
        -4.8456698884470630e-9)*t+9.2247802259858004e-9)*t
        +3.1441031864543067e-8)*t-7.2675467324291319e-8)*t
        -1.8338069950855428e-7)*t+6.5948826806917523e-7)*t
        +7.4854168574006430e-7)*t-6.1834429012694168e-6)*t
        +3.5837149798414535 7e-6)*t+4.7898783243418205 4e-5)*t
        -1.5246266466555354e-4)*t-2.6535331143276044e-5;
    y = (((((((y*t+1.8029624167359799 3e-3)*t-8.2206211541399121 5e-3)*t
        +2.4143223972444676 9e-2)*t-6.4802326694977615 2e-2)*t
        +1.0260431203219823 9e-1)*t-1.6357189552392396 9e-1)*t
        +2.2600806691662143 1e-1)*t-2.7342193149542648 2e-1)*t
        +1.4558972127503853 9e-1;
    y = 0.5*y*exp(-x*x);
    if ( x < 0.0 ) y = 1.0-y;
    return y;
}

/*************************************************************************
  Function returns primitive uniform random number in interval [0,1]
*************************************************************************/
double ran(unsigned long int *ix,
           unsigned long int *iy,
           unsigned long int *iz)
{
    double tmp;

/*  1st item of modular arithmetic  */
    *ix = (171*(*ix))%30269;
/*  2nd item of modular arithmetic  */
    *iy = (172*(*iy))%30307;
/*  3rd item of modular arithmetic  */
    *iz = (170*(*iz))%30323;
/*  Generate random number in (0,1)  */
    tmp = ((double) (*ix))/30269.0+((double) (*iy))/30307.0
          +((double) (*iz))/30323.0;
    return fmod(tmp,1.0);
}

/*************************************************************************
    Order entries of vector x[ ] in ascending order
*************************************************************************/
```

```
void heapsort( int n, double *x)
{
    int finish;
    long int i, ir, j, k;
    double tmp;

    if ( n<2 ) return;
    k = n/2;
    ir = n-1;
    finish = 0;
    while ( !finish ) {
        if ( k>0 ) {
            tmp = x[--k];
        } else {
            tmp = x[ir];
            x[ir] = x[0];
            if ( --ir==0 ) {
                x[0] = tmp;
                finish = 1;
            }
        }
        i = k;
        j = 2*k+1;
        while ( j<=ir ) {
            if ( j<ir && x[j]<x[j+1] ) j++;
            if ( tmp<x[j] ) {
                x[i] = x[j];
                i = j;
                j = 2*j+1;
            } else {
                j = ir+1;
            }
        }
        x[i] = tmp;
    }
    return;
}

/*************************************************
   Locates the minimum of the function
   func(double) to the interval [al,au].
*************************************************/
void golden( double *al, double *au, double (*func)(double) )
{
    int ;
    double r=0.618033988, vl, vu, fl, fu;

/* Count frequency of voltage in given range */
    vl = *al+r*r*(*au-*al);
    fl = func(vl);
    vu = *al+r*(*au-*al);
    fu = func(vu);
    while ( *au-*al > 5.e-5 ) {
        if ( fl >= fu ) {
            *al = vl;
            vl = vu;
            fl = fu;
            vu = *al+r*(*au-*al);
            fu = func(vu);
        } else {
            *au = vu;
```

```
            vu = vl;
            fu = fl;
            vl = *al+r*r*(*au-*al);
            fl = func(vl);
        }
    }
    return;
}

/*************************************************
    Calculates the modified Bessel function I_0(x)
*************************************************/
double bess_i0(double x)
{
    double g, t, y;
    double xvsmal=3.2e-9, xbig=7.116e+2, ybig=4.5e+307;

    t = fabs(x);
    if ( t > xbig ) return ybig;
    if ( t > 12.0 ) {
        g = exp(t-0.5*log(t));
        t = 24.0/t-1.0;
        y = -1.9567950947625728e-13;
        y = y*t+4.7229306831831040e-14;
        y = y*t+1.4457231379911B029e-12;
        y = y*t+4.3061257732B136192e-13;
        y = y*t-4.2941710672058449Ge-12;
        y = y*t-4.3462473935769108Se-12;
        y = y*t+2.8280705647555502le-12;
        y = y*t+8.2771940128604697Ge-12;
        y = y*t+1.0586362142569978Ge-11;
        y = y*t+1.8959932292080079Ae-11;
        y = y*t+4.8272663098887938Be-11;
        y = y*t+1.5614712747652883le-10;
        y = y*t+6.4799411779347205Te-10;
        y = y*t+3.4434502543142556Te-9;
        y = y*t+2.3688443405584352Be-8;
        y = y*t+2.1716050106122214Be-7;
        y = y*t+2.7977070184978559Te-6;
        y = y*t+5.5984825333737776Be-5;
        y = y*t+2.1821681721169438Ze-3;
        y = y*t+4.0107106506684741Ge-1;
        return g*y;
    } else if ( t > 4.0 ) {
        g = exp(t);
        t = 0.25*t-2.0;
        y = 2.4518525296394108Se-11;
        y = y*t-8.4690030793475489Be-11;
        y = y*t+1.2318815817541930Ze-10;
        y = y*t-3.8037017425627158Ge-10;
        y = y*t+1.5859977626817229Oe-9;
        y = y*t-4.6621548998379490Se-9;
        y = y*t+1.2413166834461642Ge-8;
        y = y*t-3.3490022193431473Be-8;
        y = y*t+8.7529183918730572Ze-8;
        y = y*t-2.1765354881644766Te-7;
        y = y*t+5.1863251906954610Se-7;
        y = y*t-1.1875284068976550Ae-6;
        y = y*t+2.6145763414226260Ae-6;
        y = y*t-5.5491776211048294Ge-6;
        y = y*t+1.1403240402174127Te-5;
```

```
        y = y*t-2.28278155280668483e-5;
        y = y*t+4.48739019580173804e-5;
        y = y*t-8.74354291104467762e-5;
        y = y*t+1.70524543267970595e-4;
        y = y*t-3.35833513200679384e-4;
        y = y*t+6.72508592273773611e-4;
        y = y*t-1.37638906941232170e-3;
        y = y*t+2.89362046530968701e-3;
        y = y*t-6.30121694459896307e-3;
        y = y*t+1.44861237337359455e-2;
        y = y*t-3.71571542566085323e-2;
        y = y*t+1.43431781856850311e-1;
        return g*y;
    } else if ( t > xvsmal ) {
        g = exp(t);
        t = 0.5*t-1.0;
        y = -7.48150165756234957e-12;
        y = y*t+4.44484466637868974e-11;
        y = y*t-2.10071360134551962e-10;
        y = y*t+1.13415934215369209e-9;
        y = y*t-5.94855273204259507e-9;
        y = y*t+2.92096168521178835e-8;
        y = y*t-1.36042013507151017e-7;
        y = y*t+6.00586861079330132e-7;
        y = y*t-2.50298976966588680e-6;
        y = y*t+9.81395862769787105e-6;
        y = y*t-3.60645571444886286e-5;
        y = y*t+1.23682594989692688e-4;
        y = y*t-3.93945320725267 20e-4;
        y = y*t+1.15888319775791686e-3;
        y = y*t-3.12923286656374358e-3;
        y = y*t+7.70061052263382555e-3;
        y = y*t-1.71317947936716536e-2;
        y = y*t+3.41505388391452167e-2;
        y = y*t-6.04316795007737183e-2;
        y = y*t+9.41616340200868389e-2;
        y = y*t-1.28895621330524993e-1;
        y = y*t+1.57686843969995904e-1;
        y = y*t-1.86478066609466760e-1;
        y = y*t+3.08508322553671039e-1;
        return g*y;
    } else {
        return 1.0;
    }
}

/*****************************************************
    Calculates the modified Bessel function I_1(x)
*****************************************************/
double bess_i1(double x)
{
    double g, t, y;
    double xvsmal=3.2e-9, xbig=7.116e+2, ybig=4.5e+307;

    t = fabs(x);
    if ( t > xbig ) return ybig;
    if ( t > 12.0 ) {
        g = exp(t-0.5*log(t));
        t = 24.0/t-1.0;
        y = 1.99488557598015486e-13;
        y = y*t-5.77176811730370560e-14;
```

```
    y = y*t-1.48765082315961139e-12;
    y = y*t-3.95353303949377536e-13;
    y = y*t+4.47736589657057690e-12;
    y = y*t+4.42966462319664383e-12;
    y = y*t-3.05957293450420224e-12;
    y = y*t-8.69631766630563635e-12;
    y = y*t-1.11795516742222899e-11;
    y = y*t-2.02947854602758139e-11;
    y = y*t-5.23524129533553498e-11;
    y = y*t-1.72060490748583241e-10;
    y = y*t-7.28107961041827952e-10;
    y = y*t-3.96757162863209348e-9;
    y = y*t-2.82537120880041703e-8;
    y = y*t-2.72684545741400871e-7;
    y = y*t-3.82795135453556215e-6;
    y = y*t-9.12475535508497109e-5;
    y = y*t-6.40545360348237412e-3;
    y = y*t+3.92624494204116555e-1;
    return g*y*(x/fabs(x));
} else if ( t > 4.0 ) {
    g = exp(t);
    t = 0.25*t-2.0;
    y = -2.27061376122617856e-11;
    y = y*t+7.79929176497056645e-11;
    y = y*t-1.10970391104678003e-10;
    y = y*t+3.38883570696523350e-10;
    y = y*t-1.41575617446629553e-9;
    y = y*t+4.11321223904934809e-9;
    y = y*t-1.07563514207617768e-8;
    y = y*t+2.84961041291017650e-8;
    y = y*t-7.28978293484163628e-8;
    y = y*t+1.76305222240064495e-7;
    y = y*t-4.05456611578551130e-7;
    y = y*t+8.86951515545183908e-7;
    y = y*t-1.83910206626348772e-6;
    y = y*t+3.60186151617732531e-6;
    y = y*t-6.63144162982509821e-6;
    y = y*t+1.13818992442463952e-5;
    y = y*t-1.79026222757948636e-5;
    y = y*t+2.47493270133518925e-5;
    y = y*t-2.62051678511418163e-5;
    y = y*t+5.21557319070236939e-6;
    y = y*t+8.47999438119288094e-5;
    y = y*t-3.67626180992174570e-4;
    y = y*t+1.17313412856965374e-3;
    y = y*t-3.40759647928956354e-3;
    y = y*t+9.76021102528646704e-3;
    y = y*t-2.99140923897405570e-2;
    y = y*t+1.34142493292698178e-1;
    return g*y*(x/fabs(x));
} else if ( t > xvsmal ) {
    t = 0.126*t*t-1.0;
    y = 6.24387910353848320e-14;
    y = y*t+4.17372709788222413e-12;
    y = y*t+2.32856921884663846e-10;
    y = y*t+1.06662712314503955e-8;
    y = y*t+3.92368710996392755e-7;
    y = y*t+1.12849795779951847e-5;
    y = y*t+2.46224314039278904e-4;
    y = y*t+3.84763940423809498e-3;
    y = y*t+4.09286371827770484e-2;
```

```
        y = y*t+2.68657659522092832e-1;
        y = y*t+9.28758890114609554e-1;
        y = y*t+1.19741654963670236;
        return x*y;
    } else {
        return 0.5*x;
    }
}

/*************************************************
    Calculates the modified Bessel function K_0(x)
*************************************************/
double bess_k0(double x)
{
    double g, t, y;
    double xvsmal=3.2e-9, egam=5.77215664901532861e-1, xbig=7.051e+2;

    if ( x <= 0.0 ) {
        printf("\nK0 evaluated for non-positive argument!");
        return 0.0;
    }
    if ( x >= xbig ) return 0.0;
    if ( x > 4.0 ) {
        t = 10.0/(1.0+x)-1.0;
        y = 4.48741979886551040e-14;
        y = y*t-1.28108310826991616e-13;
        y = y*t+2.06328892562554880e-13;
        y = y*t-7.31344482663931904e-13;
        y = y*t+2.85481235167705907e-12;
        y = y*t-1.11391758572647639e-11;
        y = y*t+3.49564293256545992e-11;
        y = y*t-2.22829582288833285e-10;
        y = y*t+1.75359321273580603e-10;
        y = y*t-9.41655321137176073e-9;
        y = y*t-4.16044811174114579e-8;
        y = y*t-7.69177622529272933e-7;
        y = y*t-6.31692398338746470e-6;
        y = y*t-9.02553345187404564e-5;
        y = y*t-9.25551464765637133e-4;
        y = y*t-1.72683652385321641e-2;
        y = y*t+1.23688664769425422;
        return y*exp(-x)/sqrt(x);
    } else if ( x > 2.0 ) {
        t = x-3.0;
        y = 2.43538242247537459e-12;
        y = y*t-7.39672783987933184e-12;
        y = y*t+9.11109430833001267e-12;
        y = y*t-2.97787564633235128e-11;
        y = y*t+1.28905587479980147e-10;
        y = y*t-4.03424607871960089e-10;
        y = y*t+1.22424982779432970e-9;
        y = y*t-3.88349705250555658e-9;
        y = y*t+1.23923137898346852e-8;
        y = y*t-3.95403255713518420e-8;
        y = y*t+1.26672629417567360e-7;
        y = y*t-4.07851207862189007e-7;
        y = y*t+1.32062261058932425e-6;
        y = y*t-4.30373871727268511e-6;
        y = y*t+1.41376609343622727e-5;
        y = y*t-4.68936653814896712e-5;
        y = y*t+1.57451516235860573e-4;
```

```
        y = y*t-5.37145622971910027e-4;
        y = y*t+1.87292939725962385e-3;
        y = y*t-6.74459607940169198e-3;
        y = y*t+2.56253646031960321e-2;
        y = y*t-1.08801882084935132e-1;
        y = y*t+6.97761598043851776e-1;
        return y*exp(-x);
    } else if ( x > 1.0 ) {
        t = 2.0*x-3.0;
        y = 2.57466288575820595e-12;
        y = y*t-7.83738609108569293e-12;
        y = y*t+9.74410152270679245e-12;
        y = y*t-3.19241059198852137e-11;
        y = y*t+1.37999268074442719e-10;
        y = y*t-4.33326665618780914e-10;
        y = y*t+1.32069362385968867e-9;
        y = y*t-4.20597329258249948e-9;
        y = y*t+1.34790467361340101e-8;
        y = y*t-4.32185089841834127e-8;
        y = y*t+1.39217270224614153e-7;
        y = y*t-4.51017292375200017e-7;
        y = y*t+1.47055796078231691e-6;
        y = y*t-4.83134250336922161e-6;
        y = y*t+1.60185974149720562e-5;
        y = y*t-5.37101208898441760e-5;
        y = y*t+1.82652460089342789e-4;
        y = y*t-6.32678357460594866e-4;
        y = y*t+2.24709729617770471e-3;
        y = y*t-8.27780350351692662e-3;
        y = y*t+3.23582010649659009e-2;
        y = y*t-1.42477910128828254e-1;
        y = y*t+9.68210053294896496e-1;
        return y*exp(-x);
    } else if ( x >  xvsmal ) {
        t = 2.0*x*x-1.0;
        g = 1.90674197614561280e-14;
        g = g*t+7.49110736894134794e-12;
        g = g*t+2.16382411824721532e-9;
        g = g*t+4.34562671546158210e-7;
        g = g*t+5.59702338227915383e-5;
        g = g*t+4.07157485171389048e-3;
        g = g*t+1.32976966478338191e-1;
        g = g*t+1.12896092945412762;
        y = 1.05407716191360000e-16;
        y = y*t+5.16867886946332160e-14;
        y = y*t+1.92405264219706684e-11;
        y = y*t+5.19906865800665633e-9;
        y = y*t+9.57878493265929443e-7;
        y = y*t+1.09534292632401542e-4;
        y = y*t+6.63513979313943827e-3;
        y = y*t+1.52436921799395196e-1;
        y = y*t+2.61841879258687055e-1;
        return -g*log(x)+y;
    } else {
        return -(log(0.5*x)+egam);
    }
}

/*************************************************
    Calculates the modified Bessel function K_1(x)
*************************************************/
```

```
double bess_k1(double x)
{
    double g, t, y;
    double xsmall=7.9e-10, xbig=7.051e+2, xsest=2.23e-308;

    if ( x <= 0.0 ) {
       printf("\nK1 evaluated for non-positive argument!");
       return 0.0;
    }
    if ( x >= xbig ) return 0.0;
    if ( x <= xsest ) {
        printf("\nK1 evaluated for very small positive argument!");
        return 1.0/xsest;
    }
    if ( x >= xbig ) return 0.0;
    if ( x > 4.0 ) {
        t = 10.0/(1.0+x)-1.0;
        y = -4.77850238111580160e-14;
        y = y*t+1.39321122940600320e-13;
        y = y*t-2.19287104441802752e-13;
        y = y*t+8.58211523713560576e-13;
        y = y*t-2.60774502020271104e-12;
        y = y*t+1.72026097285930936e-11;
        y = y*t+6.97075379117731379e-12;
        y = y*t+6.77685943857586682e-10;
        y = y*t+3.82717692121436315e-9;
        y = y*t+4.86661420008153956e-8;
        y = y*t+4.07553866931843484e-7;
        y = y*t+4.32776409784235211e-6;
        y = y*t+4.04720631528495020e-5;
        y = y*t+4.29973970898766831e-4;
        y = y*t+4.31639434283445364e-3;
        y = y*t+5.44845254318931612e-2;
        y = y*t+1.30387573604230402;
        return y*exp(-x)/sqrt(x);
    } else if ( x > 2.0 ) {
        t = x-3.0;
        y = -7.36478297050421658e-12;
        y = y*t+2.14736751065133220e-11;
        y = y*t-2.02680401514735862e-11;
        y = y*t+6.44913423545894175e-11;
        y = y*t-3.09667392343245062e-10;
        y = y*t+9.20781685906110546e-10;
        y = y*t-2.59039399308009059e-9;
        y = y*t+7.79421651144832709e-9;
        y = y*t-2.35855618461025265e-8;
        y = y*t+7.08723366696569880e-8;
        y = y*t-2.12969229346310343e-7;
        y = y*t+6.40581814037398274e-7;
        y = y*t-1.92794586996432593e-6;
        y = y*t+5.80692311842296724e-6;
        y = y*t-1.75089594354079944e-5;
        y = y*t+5.28712919123131781e-5;
        y = y*t-1.59994873621599146e-4;
        y = y*t+4.85707174778663652e-4;
        y = y*t-1.48185472032688523e-3;
        y = y*t+4.55865751206724687e-3;
        y = y*t-1.42363136684423646e-2;
        y = y*t+4.58591528414023064e-2;
        y = y*t-1.60052611291327173e-1;
        y = y*t+8.06563480128786903e-1;
```

```
        return y*exp(-x);
    } else if ( x > 1.0 ) {
        t = 2.0*x-3.0;
        y = -1.46639291782948454e-11;
        y = y*t+4.27404330568767242e-11;
        y = y*t-4.02691066627023831e-11;
        y = y*t+1.28044023949946257e-10;
        y = y*t-6.15211416898895086e-10;
        y = y*t+1.82808381381205361e-9;
        y = y*t-5.13783508140332214e-9;
        y = y*t+1.54456653909012693e-8;
        y = y*t-4.66928912168020101e-8;
        y = y*t+1.40138351985185509e-7;
        y = y*t-4.20507152338934956e-7;
        y = y*t+1.26265578331941923e-6;
        y = y*t-3.79227698821142908e-6;
        y = y*t+1.13930169202553526e-5;
        y = y*t-3.42424912211942134e-5;
        y = y*t+1.02982746700060730e-4;
        y = y*t-3.10007681013626626e-4;
        y = y*t+9.34594154387642940e-4;
        y = y*t-2.82450787841655951e-3;
        y = y*t+8.57388087067410089e-3;
        y = y*t-2.62545818729427417e-2;
        y = y*t+8.20250220860693888e-2;
        y = y*t-2.71910714388689413e-1;
        y = y*t+1.24816587355255299;
        return y*exp(-x);
    } else if ( x > xsmall ) {
        t = 2.0*x*x-1.0;
        g = 1.16964962439910400e-15;
        g = g*t+5.33888268665658944e-13;
        g = g*t+1.79784792380156752e-10;
        g = g*t+4.32764823642997753e-8;
        g = g*t+6.95300274548206237e-6;
        g = g*t+6.71642805873498653e-4;
        g = g*t+3.25725988137110495e-2;
        g = g*t+5.31907866913352762e-1;
        y = 3.29881058019865600e-15;
        y = y*t+1.40917103024514301e-12;
        y = y*t+4.46828628435618679e-10;
        y = y*t+9.96686689273781531e-8;
        y = y*t+1.44612432533006139e-5;
        y = y*t+1.20333585668219028e-3;
        y = y*t+4.50490442966943726e-2;
        y = y*t+3.51825828289325536e-1;
        return (g*log(x)-y)*x+1.0/x;
    } else {
        return 1.0/x;
    }
}
```

A.7 Bandwidth calculations

The calculation of the bandwidth is required in all problems associated with the simulation of dendritic morphology. As described in Chapter 5 there are two options for the bandwidth - regular and correlated. The functions to generate both are included here.

```
/*************************************************************
        FUNCTION RETURNS SCOTTS BANDWIDTH
*************************************************************/
double scott_bandwidth( int nobs, double *obs )
{
    extern double sigma;
    double mu, h, tmp;
    int k;

/*  STEP 1. - Calculate mean and standard deviation */
    mu = sigma = 0.0;
    for ( k=0 ; k<nobs ; k++ ) {
        mu += obs[k];
        sigma += pow(obs[k], 2);
    }
    mu /= ((double) nobs);
    sigma /= ((double) nobs);
    sigma = sqrt(sigma-mu*mu);

/*  STEP 2. - Determine mutiplying factor by dimension */
    if ( DIM == 1 ) {
        tmp = 1.06;
    } else if ( DIM == 2 ) {
        tmp = 0.96;
    } else {
        tmp = pow(((4.0/(2.0*DIM+1.0))), (1.0/(DIM+4)));
    }

/*  STEP 3. - Calculate bandwidth */
    h = tmp*sigma/pow((double) nobs, 1.0/(4.0+DIM));

    return h;
}

/*************************************************************************
        FUNCTION RETURNS THE CORRELATED SCOTTS BANDWIDTH
*************************************************************************/
void scott_correlated( int nobs, double *hx, double *hy, double *x, double *y)
{
    extern double sigmax, sigmay;
    double mux, muy, h, rho;
    int j, k;

/*  STEP 1. - Calculate mean and standard deviation of x */
    mux = sigmax = 0.0;
    for ( k=0 ; k<nobs ; k++ ) {
        mux += x[k];
        sigmax += pow(x[k], 2);
    }
    mux /= ((double) nobs);
    sigmax /= ((double) nobs);
    sigmax = sqrt(sigmax-mux*mux);
```

```
/*  STEP 2. - Calculate mean and standard deviation of y */
    muy = sigmay = 0.0;
    for ( k=0 ; k<nobs ; k++ ) {
        muy += y[k];
        sigmay += pow(y[k], 2);
    }
    muy /= ((double) nobs);
    sigmay /= ((double) nobs);
    sigmay = sqrt(sigmay-muy*muy);

/*  STEP 3. - Calculate rho */
    rho = 0.0;
    for ( k=0 ; k<nobs ; k++ ) rho += (x[k]-mux)*(y[k]-muy);
    rho /= (((double) nobs)*sigmax*sigmay);

/*  STEP 3. - Calculate bandwidth */
    *hx = sigmax*pow((1.0-rho*rho), 5.0/12.0)*pow((1.0+0.5*rho*rho),
                                -1.0/6.0)*pow(((double) nobs), -1.0/6.0);
    *hy = sigmay*pow((1.0-rho*rho), 5.0/12.0)*pow((1.0+0.5*rho*rho),
                                -1.0/6.0)*pow(((double) nobs), -1.0/6.0);

    return;
}
```

A.8 Fast Fourier Transform

This section contains the code used to perform the Fast Fourier Transform in the calculation of the propagated action potential in Chapter 2.

```
/*********************************************************
    Function to factorize n into prime factors 2, 3 and 5
*********************************************************/
int *factorize( int n )
{
    int n5=0, n3=0, n2=0, nt, *ifac;

/*  Step 1. - Check n is not zero */
    if ( n==0 ) {
        printf("Intoger n is zero - failure\n");
        return ifac;
    }

/*  Step 2. - Factor 2, 3 and 5 from n */
    nt = n;
    while ( nt%5 == 0 ) {
        n5++;
        nt /= 5;
    }
    while ( nt%3 == 0 ) {
        n3++;
        nt /= 3;
    }
    while ( nt%2 == 0 ) {
        n2++;
        nt /= 2;
    }

/*  Step 3. - Check that n is completely factorised */
    if ( nt > 1 ) {
        printf("\nInteger n has factors larger than 5 - failure\n");
        ifac = NULL;
        return ifac;
    }

/*  Step 4. - Fill vector of factors */
    ifac = (int *) malloc( (n2+n3+n5+1)*sizeof(int) );
    ifac[0] = n2+n3+n5;
    for ( nt=1 ; nt<=n5 ; nt++ ) ifac[nt] = 5;
    for ( nt=n5+1 ; nt<=n5+n3 ; nt++ ) ifac[nt] = 3;
    for ( nt=n5+n3+1 ; nt<=n5+n3+n2 ; nt++ ) ifac[nt] = 2;
    return ifac;
}

/*********************************************************
       Computes trignometric expressions needed in FFT

       trig[0][0..n-1] = cos(2.0*PI*K/N)  0 <= K <= N-1
       trig[1][0..n-1] = sin(2.0*PI*K/N)  0 <= K <= N-1
*********************************************************/
void TrigVals( int n, double **trig )
{
    double fac, angle;
    int j;
```

```
    fac = 8.0*atan(1.0)/((double) n);
    for ( j=0 ; j<n ; j++ ) {
        angle = fac*((double) j);
        trig[0][j] = cos(angle);
        trig[1][j] = sin(angle);
    }
    return;
}

void fft( int n, double **ai, double **ao)
{
    int ka=1, odd=1, i, *ifac, *factorize( int );
    void TrigVals( int, double **);
    void pass( int, double **, double **, int, int, double **);
    double *ptr, **trig;

    ifac = factorize(n);
    trig = (double **) malloc( 2*sizeof(double *) );
    trig[0] = (double *) malloc( n*sizeof(double) );
    trig[1] = (double *) malloc( n*sizeof(double) );
    TrigVals( n, trig);
    for ( i=1 ; i<=ifac[0] ; i++ ) {
        if ( odd ) {
            pass( n, ai, ao, ifac[i], ka, trig);
        } else {
            pass( n, ao, ai, ifac[i], ka, trig);
        }
        odd = !odd;
        ka *= ifac[i];
    }
    if ( odd ) {
        ptr = ao[0];
        ao[0] = ai[0];
        ai[0] = ptr;
        ptr = ao[1];
        ao[1] = ai[1];
        ai[1] = ptr;
    }
    free(ifac);
    free(trig[0]);
    free(trig[1]);
    free(trig);
    return;
}

void pass(int n,double **a,double **c,int ifac,int ka,double **trig)
{
    int ind[5], jnd[5], mval, j, k, ival, jval, jump,
        i0, i1, i2, i3, i4, j0, j1, j2, j3, j4;
    double ar, ai, br, bi, cr, ci, dr, di, er, ei, fr, fi,
           c1, c2, c3, c4, s1, s2, s3, s4;
    static double sin36=0.587785252292471, sin60=0.866025403784439,
                  sin72=0.951056516295153, fact1=0.559016994374947;

/*  Step 1. - Initialise indexing */
    mval = n/ifac;
    for ( k=0 ; k<ifac ; k++ ) {
        ind[k] = k*mval;
        jnd[k] = k*ka;
    }
```

```
    jump = (ifac-1)*ka;
    ival = jval = 0;

/*  Step 2. - Compute FFT */
    for ( k=0 ; k<=mval-ka ; k+=ka ) {
        for ( j=0 ; j<ka ; j++ ) {
            if ( ifac == 2 ) {
                i0 = ind[0]+ival;
                i1 = ind[1]+ival;
                j0 = jnd[0]+jval;
                j1 = jnd[1]+jval;
                c1 = trig[0][k];
                s1 = trig[1][k];
                if ( k==0 ) {
                    c[0][j0] = a[0][i0]+a[0][i1];
                    c[1][j0] = a[1][i0]+a[1][i1];
                    c[0][j1] = a[0][i0]-a[0][i1];
                    c[1][j1] = a[1][i0]-a[1][i1];
                } else {
                    c[0][j0] = a[0][i0]+a[0][i1];
                    c[1][j0] = a[1][i0]+a[1][i1];
                    ar = a[0][i0]-a[0][i1];
                    ai = a[1][i0]-a[1][i1];
                    c[0][j1] = c1*ar-s1*ai;
                    c[1][j1] = s1*ar+c1*ai;
                }
            } else if ( ifac == 3 ) {
                i0 = ind[0]+ival;
                i1 = ind[1]+ival;
                i2 = ind[2]+ival;
                j0 = ind[0]+jval;
                j1 = jnd[1]+jval;
                j2 = jnd[2]+jval;
                if ( k == 0 ) {
                    ar = a[0][i1]+a[0][i2];
                    ai = a[1][i1]+a[1][i2];
                    c[0][j0] = a[0][i0]+ar;
                    c[1][j0] = a[1][i0]+ai;
                    ar = a[0][i0]-0.5*ar;
                    ai = a[1][i0]-0.5*ai;
                    br = sin60*(a[0][i1]-a[0][i2]);
                    bi = sin60*(a[1][i1]-a[1][i2]);
                    c[0][j1] = ar-bi;
                    c[1][j1] = ai+br;
                    c[0][j2] = ar+bi;
                    c[1][j2] = ai-br;
                } else {
                    c1 = trig[0][k];
                    c2 = trig[0][2*k];
                    s1 = trig[1][k];
                    s2 = trig[1][2*k];
                    ar = a[0][i1]+a[0][i2];
                    ai = a[1][i1]+a[1][i2];
                    c[0][j0] = a[0][i0]+ar;
                    c[1][j0] = a[1][i0]+ai;
                    ar = a[0][i0]-0.5*ar;
                    ai = a[1][i0]-0.5*ai;
                    br = sin60*(a[0][i1]-a[0][i2]);
                    bi = sin60*(a[1][i1]-a[1][i2]);
                    cr = ar-bi;
                    ci = ai+br;
```

```
            c[0][j1] = c1*cr-s1*ci;
            c[1][j1] = s1*cr+c1*ci;
            cr = ar+bi;
            ci = ai-br;
            c[0][j2] = c2*cr-s2*ci;
            c[1][j2] = s2*cr+c2*ci;
        }
    } else if ( ifac == 5 ) {
        i0 = ind[0]+ival;
        i1 = ind[1]+ival;
        i2 = ind[2]+ival;
        i3 = ind[3]+ival;
        i4 = ind[4]+ival;
        j0 = jnd[0]+jval;
        j1 = jnd[1]+jval;
        j2 = jnd[2]+jval;
        j3 = jnd[3]+jval;
        j4 = jnd[4]+jval;
        if ( k == 0 ) {
            ar = a[0][i1]+a[0][i4];
            ai = a[1][i1]+a[1][i4];
            br = a[0][i2]+a[0][i3];
            bi = a[1][i2]+a[1][i3];
            cr = ar+br;
            ci = ai+bi;
            c[0][j0] = a[0][i0]+cr;
            c[1][j0] = a[1][i0]+ci;
            br = fact1*(ar-br);
            bi = fact1*(ai-bi);
            cr = a[0][i0]-0.25*cr;
            ci = a[1][i0]-0.25*ci;
            ar = cr+br;
            ai = ci+bi;
            br = cr-br;
            bi = ci-bi;
            cr = a[0][i1]-a[0][i4];
            ci = a[1][i1]-a[1][i4];
            dr = a[0][i2]-a[0][i3];
            di = a[1][i2]-a[1][i3];
            er = sin72*cr+sin36*dr;
            ei = sin72*ci+sin36*di;
            c[0][j1] = ar-ei;
            c[1][j1] = ai+er;
            c[0][j4] = ar+ei;
            c[1][j4] = ai-er;
            er = sin36*cr-sin72*dr;
            ei = sin36*ci-sin72*di;
            c[0][j2] = br-ei;
            c[1][j2] = bi+er;
            c[0][j3] = br+ei;
            c[1][j3] = bi-er;
        } else {
            c1 = trig[0][k];
            c2 = trig[0][2*k];
            c3 = trig[0][3*k];
            c4 = trig[0][4*k];
            s1 = trig[1][k];
            s2 = trig[1][2*k];
            s3 = trig[1][3*k];
            s4 = trig[1][4*k];
            ar = a[0][i1]+a[0][i4];
```

```
                    ai = a[1][i1]+a[1][i4];
                    br = a[0][i2]+a[0][i3];
                    bi = a[1][i2]+a[1][i3];
                    cr = ar+br;
                    ci = ai+bi;
                    c[0][j0] = a[0][i0]+cr;
                    c[1][j0] = a[1][i0]+ci;
                    br = fact1*(ar-br);
                    bi = fact1*(ai-bi);
                    cr = a[0][i0]-0.25*cr;
                    ci = a[1][i0]-0.25*ci;
                    ar = cr+br;
                    ai = ci+bi;
                    br = cr-br;
                    bi = ci-bi;
                    cr = a[0][i1]-a[0][i4];
                    ci = a[1][i1]-a[1][i4];
                    dr = a[0][i2]-a[0][i3];
                    di = a[1][i2]-a[1][i3];
                    er = sin72*cr+sin36*dr;
                    ei = sin72*ci+sin36*di;
                    c[0][j1] = ar-ei;
                    c[1][j1] = ai+er;
                    c[0][j4] = ar+ei;
                    c[1][j4] = ai-er;
                    fr = ar-ei;
                    fi = ai+er;
                    c[0][j1] = c1*fr-s1*fi;
                    c[1][j1] = s1*fr+c1*fi;
                    fr = ar+ei;
                    fi = ai-er;
                    c[0][j4] = c4*fr-s4*fi;
                    c[1][j4] = s4*fr+c4*fi;
                    er = sin36*cr-sin72*dr;
                    ei = sin36*ci-sin72*di;
                    fr = br-ei;
                    fi = bi+er;
                    c[0][j2] = c2*fr-s2*fi;
                    c[1][j2] = s2*fr+c2*fi;
                    fr = br+ei;
                    fi = bi-er;
                    c[0][j3] = c3*fr-s3*fi;
                    c[1][j3] = s3*fr+c3*fi;
                }
            }
            ival++;
            jval++;
        }
        jval += jump;
    }
    return;
}

/*****************************************************************
   Function takes as input the values of u(x) at the unipoints

   Input
   -----
              ur[k] --> value of u at x[k]   ( 0<= k <= N-1 )

   Output
```

```
    ------
              uc[0] --> u[0] (guaranteed to be real)
              uc[1] --> u[-nh] (guaranteed to be real)
             uc[2k] --> real part of u[k] 1<=k<=nh-1
           uc[2k+1] --> imag part of u[k] 1<=k<=nh-1
  ************************************************************/
void real_c( int n, double *ur, double *uc)
{
    double fac, theta, angle, cc, ss, s1, s2, **a, **c;
    int k, nh, nd, kk;
    void fft( int, double **, double **);

/*  Step 1. - Allocate a[ ][ ] and c[ ][ ] */
    nh = n/2;
    a = (double **) malloc( 2*sizeof(double *) );
    c = (double **) malloc( 2*sizeof(double *) );
    for ( k=0 ; k<2 ; k++ ) {
        a[k] = (double *) malloc( nh*sizeof(double) );
        c[k] = (double *) malloc( nh*sizeof(double) );
    }

/*  Step 2. - Assign a[ ][ ] */
    for ( k=0 ; k<nh ; k++ ) {
        kk = 2*k;
        a[0][k] = ur[kk];
        a[1][k] = ur[kk+1];
    }

/*  Step 3. - Apply FFT */
    fft( nh, a, c);

/*  Step 4. - Interpret c[ ][ ] to get uc[ ] */
    uc[0] = (c[0][0]+c[1][0])/((double) n);
    uc[1] = (c[0][0]-c[1][0])/((double) n);
    theta = 4.0*atan(1.0)/((double) nh);
    for ( k=1 ; k<nh ; k++ ) {
        angle = theta*((double) k);
        ss = sin(angle);
        cc = cos(angle);
        nd = nh-k;

        s1 = c[0][k]-c[0][nd];
        s2 = c[1][k]+c[1][nd];

        uc[2*k] = c[0][k]+c[0][nd]+s1*ss+s2*cc;
        uc[2*k+1] = c[1][nd]-c[1][k]+s1*cc-s2*ss;
    }
    fac = 0.5/((double) n);
    for ( k=2 ; k<n ; k++ ) uc[k] *= fac;

/*  Step 5. - Free memory */
    for ( k=0 ; k<2 ; k++ ) {
        free(a[k]);
        free(c[k]);
    }
    free(a);
    free(c);
    return;
}

 /************************************************************
```

```
   Input
   -----       ur[0] --> value of f[0] (guaranteed real)
               ur[1] --> value of f[-N/2] (guaranteed real)
             ur[2k] --> value of Re(f[k]) ( 1 <= k <= N/2-1 )
           ur[2k+1] --> value of Im(f[k]) ( 1 <= k <= N/2-1 )

   Output
   ------      uc[k] --> value of u at x[k] ( 0 <= k <= N-1 )
  ************************************************************/
void real_v( int n, double *ur, double *uc)
{
    double theta, angle, cc, ss, s1, s2, **a, **c;
    int k, nh, nd, nm, kk;
    void fft( int, double **, double **);

/*  Step 1. - Allocate a[ ][ ] and c[ ][ ] */
    nh = n/2;
    a = (double **) malloc( 2*sizeof(double *) );
    c = (double **) malloc( 2*sizeof(double *) );
    for ( k=0 ; k<2 ; k++ ) {
        a[k] = (double *) malloc( nh*sizeof(double) );
        c[k] = (double *) malloc( nh*sizeof(double) );
    }

/*  Step 2. - Assign uc[ ][ ] and ur[ ][ ] */
    c[0][0] = ur[0]+ur[1];
    c[1][0] = ur[0]-ur[1];
    theta = 4.0*atan(1.0)/((double) nh);
    for ( k=1 ; k<nh ; k ++ ) {
        kk = 2*k;
        nd = n-kk;
        s1 = ur[kk]-ur[nd];
        s2 = ur[kk+1]+ur[nd+1];
        angle = theta*((double) k);
        cc = cos(angle);
        ss = sin(angle);
        c[0][k] = ur[kk]+ur[nd]+s1*ss-s2*cc;
        c[1][k] = ur[kk+1]-ur[nd+1]+s1*cc+s2*ss;
    }

/*  Step 3. - Apply FFT */
    fft( nh, c, a);

/*  Step 4. - Interpret a[ ][ ] to get uc[ ] */
    uc[0] = a[0][0];
    uc[n-1] = a[1][0];
    for ( k=1 ; k<nh ; k++ ) {
        kk = 2*k;
        uc[kk] = a[0][k];
        uc[kk-1] = a[1][k];
    }

/*  Step 5. - Free memory */
    for ( k=0 ; k<2 ; k++ ) {
        free(a[k]);
        free(c[k]);
    }
    free(a);
    free(c);
    return;
}
```

A.9 Differential equation solver

This section contains the code used to solve the differential equations in the calculation of the propagated action potential in Chapter 2.

```
/* Global function declarations */
void intrp( int, double, double *, double, double *, double *, int,
            double **, double *);
void step( int, double *, double *, double *, double *, double *,
           double *, int *, int *, int *, int *, double **, double *,
           double *, double *, double, double, void (*fcn)(double,
           double *, double *));

 /************************************************************************
   SGSOLVE is a C translation of the FORTRAN program DE which is
   completely explained and documented in the text COMPUTER SOLUTION
   OF ORDINARY DIFFERENTIAL EQUATIONS: THE INITIAL VALUE PROBLEM
   BY L. F. SHAMPINE AND M. K. GORDON.

   SGSOLVE integrates a system of first order differential equations

              DY(I)/DT = F(T, Y(1),Y(2), ... ,Y(N))
              Y(I) GIVEN AT T

   for arbitrary order N. Initial conditions entered through y[ ] at
   "tin" are integrated to "tout" in accordance with the relative
   error "relerr" and absolute error "abserr" and output through y[ ].
   On successful output, "tin" takes the value "tout" and "ifail=1".
   SGSOLVE may be repeated as necessary provide "ifail=+2/-2" on output.
   Otherwise, "tin" contains the limit of integration prior to failure.

   SGSOLVE uses an INTEGRATOR code and an INTERPOLATION code. The
   former is based on a modified divided difference form of the ADAMS
   PECE formulae and local EXTRAPOLATION. ORDER and STEP SIZE control
   local error. Normally each application of the integrator advances
   the solution one step towards "tout". For reasons of efficiency,
   internal integration proceeds beyond "tout" though never beyond
   "tnow+10*(tout-tnow)". The latter interpolates the solution at
   "tout". If integration beyond "tout" is impossible, "ifail=-1" on
   entry.

   INPUT to SGSOLVE

   The differential equation is supplied through a void function, e.g.

         void fprime( double t, double *y, double *dy).

   The address of fprime is passed to SGSOLVE via the void pointer
   "fcn". All parameters of SGSOLVE must be suitably initialised with
   either ifail=+1, if integration beyond "tout" is possible, or
   ifail=-1 if integration beyond "tout" is impossible.

   OUTPUT from SGSOLVE

   "tin" contains the last point for which integration was successful
   - "tout" for a normal exit - and y[ ] contains the solution vector
   at "tin". The tolerances "relerr" and "abserr" are normally
   unchanged on exit except when "ifail=3" in which case they are
   increased. The error indicator on exit takes the values
```

```
   ifail = 2 -- Normal return. Integration reached "tout".
         = 3 -- Integration failed to reach "tout" because "relerr"
                and "abserr" are too small - "relerr" and "abserr"
                increased appropriately so that integration can be
                continued.
         = 4 -- Too many integration steps needed to reach "tout".
         = 5 -- Integration failed to reach "tout" - equations seem
                STIFF.
         = 6 -- Invalid input parameters (fatal error).
  ************************************************************/
void sgsolve( int n,            /* Order of system */
              int *begin,       /* Memory allocation flag */
              double *relerr,   /* Relative (local) error tolerance */
              double *abserr,   /* Absolute (local) error tolerance */
              double *tin,      /* Entry value of independent variable*/
              double tout,      /* Exit value of independent variable */
              double *y,        /* Sol'n vector of dependent variables */
              void (*fcn)( double, double *, double *),
                                /* Pointer to derivatives dy[i]/dt */
              int *ifail        /* Error indicator */ )
{
    int i, k, m, start, crash, stiff, iflag, isn, kle4, finish;
    static int maxiter=50000, isnold, kold, nostep;
    static double told, hold, delsgn;
    static double *psi, *yy, *wt, **phi, *p, *yp, *ypout;
    double x, rnderr, two_rnderr, four_rnderr, eps, del, absdel, tend,
           releps, abseps, h, min;

/* Step 0. - Allocate memory */
    if ( *begin ) {
        if ( psi ) free(psi);
        psi = (double *) malloc ( 12*sizeof(double) );
        if ( yy ) free(yy);
        yy = (double *) malloc ( n*sizeof(double) );
        if ( wt ) free(wt);
        wt = (double *) malloc ( n*sizeof(double) );
        if ( yp ) free(yp);
        yp = (double *) malloc ( n*sizeof(double) );
        if ( ypout ) free(ypout);
        ypout = (double *) malloc ( n*sizeof(double) );
        if ( p ) free(p);
        p = (double *) malloc ( n*sizeof(double) );
        if ( phi ) {
            for ( i=0 ; i<n ; i++ ) free(phi[i]);
            free(phi);
        }
        phi = (double **) malloc ( n*sizeof(double *) );
        for (i=0;i<n;i++) phi[i]=(double *) malloc(16*sizeof(double));
        *begin = 0;
    }

/* Step 1. - Determine machine precision "rnderr" */
    rnderr = 1.0;
    while ( rnderr+1.0 != 1.0 ) rnderr *= 0.5;
    two_rnderr = 4.0*rnderr;
    four_rnderr = 8.0*rnderr;

/* Step 2. - Test for invalid entry parameters */
    if ( n<1 ) *ifail = 6;                  /* Order not set */
    if ( *tin==tout ) *ifail = 6;           /* Zero int of integration */
    if ( *relerr<0.0 || *abserr<0.0 ) *ifail = 6;   /* At least one
```

```
                                                tolerance not set */
    eps = ( *relerr>=*abserr ) ? *relerr : *abserr;
    if ( eps<=0.0 ) *ifail = 6;          /* No positive tolerance set */
    if ( *ifail==0 ) *ifail = 6;         /* Error indicator not set */
    isn = ( *ifail<0 ) ? -1 : 1;
    iflag = isn*(*ifail);
    if ( iflag>=2 && *tin!=told ) *ifail = 6;
                                         /* Point of re-entry changed */
    if ( *ifail==6 ) return;

/* Step 3. - Set interval of integration and initialise step counter.
    Adjust input error tolerances to define weight vector for
    function STEP */
    finish = 0;
    del = tout-(*tin);
    absdel = fabs(del);
    tend = (*tin)+10.0*del;
    if ( isn<0 ) tend = tout;
    nostep = 0;
    kle4 = 0;
    stiff = 0;
    releps = (*relerr)/eps;
    abseps = (*abserr)/eps;

/* Step 4. - On a start/restart, set work variables "x" and yy[ ], store
    direction of integration and initialise step size. */
    if ( iflag==1 || isnold<0 || delsgn*del<=0.0 ) {
        start = 1;
        x = *tin;
        for ( m=0 ; m<n ; m++ ) yy[m] = y[m];
        delsgn = ( del>=0.0 ) ? 1.0 : -1.0;
        h = four_rnderr*fabs(x);
        if ( fabs(tout-x) > h ) h = fabs(tout-x);
        if ( tout < x ) h = -h;
    }
    while ( !finish ) {
        finish = 1;
        if ( fabs(x-(*tin)) >= absdel ) {
/* Step 5. - Already beyond output point and so interpolate and return */
            intrp( n, x, yy, tout, y, ypout, kold, phi, psi);
            *ifail = 2;
            *tin = tout;
            isnold = isn;
        } else if ( isn<0 && fabs(tout-x)<four_rnderr*fabs(x) ) {
/* Step 6. - No passage beyond "tout" but close enough to extrapolate */
            h = tout-x;
            (*fcn)( x, yy, yp);
            for ( m=0 ; m<n ; m++ ) y[m] = yy[m]+h*yp[m];
            *ifail = 2;
            *tin = tout;
            isnold = isn;
        } else if ( nostep >= maxitor ) {
/* Step 7. - Test for too much work */
            *ifail = 4*isn;
            if ( stiff ) *ifail = 5*isn;
            for ( m=0 ; m<n ; m++ ) y[m] = yy[m];
            *tin = x;
            isnold = 1;
        } else {
/* Step 8. - Limit step size, set weight vector and take step */
            min = fabs(tend-x);
```

```
            if ( fabs(h) < min ) min = fabs(h);
            h = ( h >= 0.0 ) ? min : -min;
            for ( m=0 ; m<n ; m++ ) wt[m] = releps*fabs(yy[m])+abseps;
            step( n, &x, yy, &h, &eps, wt, &hold, &start, &crash, &k,
                  &kold, phi, p, yp, psi, two_rnderr, four_rnderr, fcn);
/* Step 9. - Test for tolerances that are too small */
            if ( crash ) {
                *ifail = 3*isn;
                *relerr = eps*releps;
                *abserr = eps*abseps;
                for ( m=0 ; m<n ; m++ ) y[m] = yy[m];
                *tin = x;
                isnold = 1;
            } else {
/* Step 10. - Increase counter and test for stiffness */
                nostep++;
                kle4++;
                if ( kold>4 ) kle4 = 0;
                if ( kle4>=50 ) stiff = 1;
                finish = 0;
            }
        }
    }
    told = *tin;
/*    if ( *reset_mem==1 ) {
        begin = 1;
        *reset_mem = 0;
        free(psi);
        free(yy);
        free(wt);
        free(yp);
        free(ypout);
        free(p);
        for ( i=0 ; i<n ; i++ ) free(phi[i]);
        free(phi);
    } */
    return;
}

/*********************************************************************
   This code is the C translation of FORTRAN interpolation code which
   is completely explained and documented in the text COMPUTER
   SOLUTION OF ORDINARY DIFFERENTIAL EQUATIONS: THE INITIAL VALUE
   PROBLEM BY L. F. SHAMPINE AND M. K. GORDON.

   Routine STEP approximates the solution near "x" by a polynomial.
   INTRP approximates the solution at "xout" by evaluating the
   polynomial there. Information defining this polynomial is passed
   from STEP and so INTRP cannot be used as a stand-alone routine.

   INPUT to INTRP

      Parameters are pssed from STEP in the normal way.

   OUTPUT from INTRP

      yout[ ] -- Solution vector at "xout"
     ypout[ ] -- Derivative of solution at "xout"
*********************************************************************/
void intrp( int n,          /* Order of system */
            double x,       /* Point where y[ ] is approx'd by STEP */
```

```
            double *y,     /* Solution vector at "x" */
            double xout,   /* Point at which solution is required */
            double *yout,  /* Solution vector at "xout" */
            double *ypout, /* Derivative vector at "xout" */
            int kold,      /* Order of scheme */
            double **phi,  /* Matrix of workspace */
            double *psi    /* Vector of workspace */ )
{
    int i, j, jm1, m, lim;
    double hi, term, gama, eta, tmp1, tmp2, *g, *w, *r;

/* Step 0. - Allocate memory to g[ ], w[ ] and rho[ ] */
    g = (double *) malloc( 13*sizeof(double) );
    w = (double *) malloc( 13*sizeof(double) );
    r = (double *) malloc( 13*sizeof(double) );

/* Step 1. - Initialise g[0] and r[0] */
    g[0] = 1.0;
    r[0] = 1.0;
    hi = xout-x;

/* Step 2. - Initialise w[ ] for computing g[ ] */
    for ( i=0 ; i<=kold ; i++ ) w[i] = 1.0/((double) i+1);

/* Step 3. - Compute g[ ] */
    for ( term=0.0,j=1 ; j<=kold ; j++ ) {
        jm1 = j-1;
        tmp1 = 1.0/psi[jm1];
        gama = tmp1*(hi+term);
        eta = tmp1*hi;
        lim = kold-j;
        for ( i=0 ; i<=lim ; i++ ) w[i] = gama*w[i]-eta*w[i+1];
        g[j] = w[0];
        r[j] = gama*r[jm1];
        term = psi[jm1];
    }

/* Step 3. - Interpolate */
    for ( m=0 ; m<n ; m++ ) {
        ypout[m] = 0.0;
        yout[m] = 0.0;
    }
    for ( j=0 ; j<=kold ; j++ ) {
        i = kold-j;
        tmp1 = g[i];
        tmp2 = r[i];
        for ( m=0 ; m<n ; m++ ) {
            yout[m] += tmp1*phi[m][i];
            ypout[m] += tmp2*phi[m][i];
        }
    }
    for ( m=0 ; m<n ; m++ ) yout[m] = y[m]+hi*yout[m];
    free(g);
    free(w);
    free(r);
    return;
}

 /***********************************************************
   This routine is a C translation of the FORTRAN STEP routine which
   is completely explained and documented in the text COMPUTER
```

```
SOLUTION OF ORDINARY DIFFERENTIAL EQUATIONS: THE INITIAL VALUE
PROBLEM BY L. F. SHAMPINE AND M. K. GORDON.

STEP integrates a system of first order ordinary differential
equations from "x" to "x+h" using a modified divided difference
form of the ADAMS PECE formulas. Local extrapolation is used to
improve absolute stability and accuracy. Order and step-size are
adjusted to control local error. Special devices control roundoff
error and detect over ambitious accuracy requests.

PARAMETER DEFINITIONS

    x -- Independent variable
 y[ ] -- Solution vector at "x"
yp[ ] -- Derivative of solution vector at "x" after a successful step
    n -- Order of system
    h -- Appropriate step-size for next step. Determined by code
  eps -- Local error tolerance
wt[ ] -- Vector of weights for error criterion
start -- Set 1 (TRUE) for first step and set 0 (FALSE) otherwise
 hold -- Step size for last successful step
    k -- Appropriate order for next step. Determined by code.
 kold -- Order used for last successful step
crash -- Set 1 (TRUE) when no step possible and set 0 (FALSE)
         otherwise

The arrays phi[ ][ ] and psi[ ] are needed for the (interpolation)
function INTRP and p[ ] is internal. The system of differential
equations DY(I)/DT = F(T, Y(1),Y(2), ... ,Y(N)) is supplied through
a void function, e.g.

       void fprime( double t, double *y, double *dy).

The address of fprime is passed to STEP via the void pointer "fcn".

INPUT to STEP (first call) ... SET

    x -- Initial value of independent variable
 y[ ] -- Initial value of solution vector at "x"
    n -- Order of system
    h -- Maximum step-size indicating direction of integration
  eps -- Local error tolerance per step
wt[ ] -- Vector of weights for error criterion
start -- Set 1 (TRUE) for first step and set 0 (FALSE) otherwise

STEP needs the L2 NORM of the vector with components
local_error[j]/wt[j] to be less than "eps" for a successful step.
The array wt[ ] allows the specification of different error tests
in accordance with the criteria

  wt[j] = 1.0 specifies absolute error,
        = fabs(y[j]) specifies error relative to the most recent
          value of y[j], the j-th component of the solution vector,
        = fabs(yp[j]) specifies error relative to the most recent
          value of yp[j], the j-th component of the derivative of
          the solution vector,
        = MAX(wt[j],fabs(y[j])) specifies error relative to the
          largest magnitude of the j-th component obtained so far,
        = fabs(y[j])*relerr/eps+abserr/eps specifies a mixed
          relative/absolute error test where relerr/abserr are
          relative/absolute error and "eps" is max(relerr,abserr).
```

```
   INPUT to STEP (subsequent calls) ... SET

   STEP returns all that is needed to continue integration, including
   step-size "h" and order "k". With the exception of step-size, the
   error tolerance and the weights, none of the parameters should be
   changed. Array wt[ ] must be updated after each step to maintain
   relative error tests. Normally integration is continued just beyond
   the desired endpoint and INTRP used to interpolate solution. If it
   is not possible to integrate beyond the endpoint, "h" is
   adjusted to meet "tout". To change the direction of integration,
   set "start=1" (TRUE) before calling STEP again. This is the only
   situation in which "start" should be changed.

   OUTPUT from STEP (successful)

   After each successful step, "start=crash=0". The independent variable
   "x" is advanced by "hold" from its value on entry. The solution vector
   y[ ] is given at the new value of "x" while all other parameters
   contain information needed to continue integration from the new
   "x".

   OUTPUT from STEP (unsuccessful)

   If the error tolerance is too small for machine precision, "crash=1"
   and no step is taken. An appropriate step-size and error tolerance for
   continuation are estimated and all other parameters re-instated before
   returning. To continue with the new tolerance, call code again. A
   restart is neither required nor desirable.
  ***************************************************************/
void step( int n,                  /* Order of first order system */
           double *x,              /* Current value of independent variable*/
           double *y,              /* The solution vector */
           double *h,              /* The suggested step-size */
           double *eps,            /* Max of the rel/abs error tolerances */
           double *wt,             /* Weights for error tolerances */
           double *hold,           /* Last successful step-size */
           int *start,             /* Initialisation flag */
           int *crash,             /* Failed step indicator */
           int *k,                 /* Suggested order for the next step */
           int *kold,              /* Order of last successful step */
           double **phi,           /* Workspace */
           double *p,              /* Workspace */
           double *yp,             /* Holds derivative of solution vector */
           double *psi,            /* Workspace */
           double two_rnderr,      /* Two times the machine precision */
           double four_rnderr,     /* Four times the machine precision */
           void (*fcn)( double, double *, double *)
                                   /* Pointer to derivatives dy[i]/dt */ )
{
    int i, j, m, km3, km2, km1, kp1, nsm1, nspi, im1, lim, ipi, nfail;
    static int phasei, nornd, ns, knew;
    double round, halfeps, sum, absh, tmp, tmp1, tmp2, erk, erkm1,
           erkm2, hnew, hmin, xold, err, erkp1;
    static double alfa[12], beta[12], w[12], v[12], g[13], sig[13];
    static double gstr[13]={0.5000, 0.0833, 0.0417, 0.0264, 0.0188,
                            0.0143, 0.0114, 0.00936, 0.00789, 0.00679,
                            0.00592, 0.00524, 0.00468};
    static double two[13]={ 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0,
                            256.0, 512.0, 1024.0, 2048.0, 4096.0, 8192.0};
```

```
/* Initialise some elements */
    g[0] = 1.0;
    g[1] = 0.5;
    sig[0] = 1.0;

/* Begin BLOCK 0
   (a) Check step-size and error tolerance not too small for rounding
       error, (b) If first step, initialise phi[ ][ ] and estimate initial
       step-size */

/* If step-size too small, set "h" to minimum step-size */
    *crash = 1;
    hmin = four_rnderr*fabs(*x);
    if ( fabs(*h) < hmin ) {
        *h = ( *h >= 0.0 ) ? hmin : -hmin;
        return;
    }
    halfeps = 0.5*(*eps);

/* If error tolerance too small, increase it to a suitable value */
    for ( round=0.0,m=0 ; m<n ; m++ ) round += pow(y[m]/wt[m],2);
    round = two_rnderr*sqrt(round);
    if ( halfeps < round ) {
        *eps = 2.0*round*(1.0+four_rnderr);
        return;
    }

/* Initialise phi[ ][ ] and determine a suitable first step */
    *crash = 0;
    if ( *start ) {
        (*fcn)( *x, y, yp);
        for ( sum=0.0,m=0 ; m<n ; m++ ) {
            phi[m][0] = yp[m];
            phi[m][1] = 0.0;
            sum += pow(yp[m]/wt[m],2);
        }
        sum = sqrt(sum);
        absh = fabs(*h);
        if ( *eps < 16.0*sum*(*h)*(*h) ) absh = 0.25*sqrt(*eps/sum);
        tmp = ( absh >= hmin ) ? absh : hmin;
        *h = ( *h >= 0.0 ) ? tmp : -tmp;
        *hold = 0.0;
        *k = 1;
        *kold = 0;
        *start = 0;
        phase1 = 1;
        nornd = 1;
        if ( halfeps <= 100.0*round ) {
            nornd = 0;
            for ( m=0 ; m<n ; m++ ) phi[m][14] = 0.0;
        }
    }

/* End BLOCK 0 and begin BLOCK 1 */
/* Compute coefficients of formulas for this step. Avoid computing
   quantities not changed when step-size remains unchanged */
    nfail = 0;
    for ( ; ; ) {
        kpi = *k+1;
        km1 = *k-1;
        km2 = *k-2;
```

```
        km3 = *k-3;

/* Integer "ns" counts the number of steps with size "h", including
   the current step. When k<=ns, no coefficients change */
        if ( *h != *hold ) ns = -1;
        ns = ( ns < *kold ) ? ns+1 : *kold;
        nsp1 = ns+1;

/* Check if alfa[ ], beta[ ], psi[ ], sig[ ] change and make them
   where necessary */
        if ( km1 >= ns ) {
            beta[ns] = 1.0;
            tmp = ((double) ns+1);
            alfa[ns] = 1.0/tmp;
            tmp1 = (*h)*tmp;
            sig[nsp1] = 1.0;
            for ( i=nsp1 ; i<=km1 ; i++ ) {
                im1 = i-1;
                tmp2 = psi[im1];
                psi[im1] = tmp1;
                beta[i] = beta[im1]*psi[im1]/tmp2;
                tmp1 = tmp2+(*h);
                alfa[i] = (*h)/tmp1;
                sig[i+1] = alfa[i]*sig[i]*((double) i+1);
            }
            psi[km1] = tmp1;

/* Compute coefficients g[ ], initialise v[ ] an set w[ ].
   g[1] is set previously */
            if ( ns > 0 ) {
                if ( *k > *kold ) {
                /* Order increased -> update diagonal part of v[ ] */
                    v[km1] = 1.0/((double) kp1*(*k) );
                    nsm1 = ns-1;
                    for ( j=1 ; j<=nsm1 ; j++ ) {
                        i = km1-j;
                        v[i] = v[i]-alfa[j]*v[i+1];
                    }
                }

/* Update v[ ] and set w[ ] */
                lim = km1-ns;
                for ( i=0 ; i<=lim ; i++ ) {
                    v[i] -= alfa[ns]*v[i+1];
                    w[i] = v[i];
                }
                g[nsp1] = w[0];
            } else {
                for ( i=0 ; i<=km1 ; i++ ) {
                    v[i] = 1.0/((double) (i+1)*(i+2));
                    w[i] = v[i];
                }
            }

/* Compute g[ ] in the work vector w[ ] */
            for ( i=nsp1 ; i<=km1 ; i++ ) {
                lim = km1-i;
                for ( j=0 ; j<=lim ; j++ ) w[j] -= alfa[i]*w[j+1];
                g[i+1] = w[0];
            }
        }
```

```
/* End BLOCK 1 and begin BLOCK 2 */
/* Predict solution p[ ]. Evaluate derivatives using predicted
   solution. Estimate local error at order "k" and errors at orders
   "k", "km1" and "km2" as if constant step size were used. Change
   phi[ ] to phi*[ ] */
        for ( i=nsp1 ; i<=km1 ; i++ ) {
            for ( m=0 ; m<n ; m++ ) phi[m][i] *= beta[i];
        }

/* Predict solution and differences */
        for ( m=0 ; m<n ; m++ ) {
            phi[m][kp1] = phi[m][*k];
            phi[m][*k] = 0.0;
            p[m] = 0.0;
        }
        for ( j=0 ; j<=km1 ; j++ ) {
            i = km1-j;
            ip1 = i+1;
            for ( m=0 ; m<n ; m++ ) {
                p[m] += g[i]*phi[m][i];
                phi[m][i] += phi[m][ip1];
            }
        }
        if ( nornd ) {
            for ( m=0 ; m<n ; m++ ) p[m] = y[m]+(*h)*p[m];
        } else {
            for ( m=0 ; m<n ; m++ ) {
                tmp = (*h)*p[m]-phi[m][14];
                p[m] = y[m]+tmp;
                phi[m][15] = (p[m]-y[m])-tmp;
            }
        }
        xold = *x;
        *x += *h;
        absh = fabs(*h);
        (*fcn)( *x, p, yp);

/* Estimate errors at orders "k", "km1" and "km2" */
        for ( erkm2=0.0,erkm1=0.0,erk=0.0,m=0 ; m<n ; m++ ) {
            tmp1 = 1.0/wt[m];
            tmp2 = yp[m]-phi[m][0];
            if ( *k > 2 ) erkm2 += pow((phi[m][km2]+tmp2)*tmp1,2);
            if ( *k > 1 ) erkm1 += pow((phi[m][km1]+tmp2)*tmp1,2);
            erk += pow(tmp2*tmp1,2);
        }
        if ( *k > 2 ) erkm2 = absh*sig[km2]*gstr[km3]*sqrt(erkm2);
        if ( *k > 1 ) erkm1 = absh*sig[km1]*gstr[km2]*sqrt(erkm1);
        tmp = absh*sqrt(erk);
        err = tmp*(g[km1]-g[*k]);
        erk = tmp*sig[*k]*gstr[km1];
        knew = *k;

/* Test if order should be reduced */
        if ( *k > 2 && erkm1 <= erk && erkm2 <= erk ) knew = km1;
        if ( *k > 1 && erkm1 <= 0.5*erk ) knew = km1;

/* End BLOCK 2 and begin BLOCK 3 */

/* The step has been unsuccessful and so restore "x", phi[ ][ ],
   psi[ ]. If this is a third consecutive failure, set order to 1. If
```

```
   step fails more than 3 times, consider an optimal step-size. Double
   error tolerance and return if estimated step-size is too small for
   machine precision. Restore "x", phi[ ][ ] and psi[ ] */
        if ( err > *eps ) {
            phase1 = 0;
            for ( *x=xold,i=0 ; i<=km1 ; i++ ) {
                tmp = 1.0/beta[i];
                ip1 = i+1;
                for ( m=0 ; m<n ; m++ )
                        phi[m][i] = tmp*(phi[m][i]-phi[m][ip1]);
            }
            for ( i=0 ; i<=km2 ; i++ ) psi[i] = psi[i+1]-(*h);

/* On 3rd failure, set order to 1. Thereafter use optimal step-size */
            nfail++;
            tmp = 0.5;
            if ( nfail>3 && halfeps<0.25*erk ) tmp=sqrt(halfeps/erk);
            if ( nfail > 2 ) knew = 1;
            *h *= tmp;
            *k = knew;
            if ( fabs(*h) < hmin ) {
                *crash = 1;
                *h = ( *h >= 0.0 ) ? hmin : -hmin;
                *eps *= 2.0;
                return;
            }
        } else {

/* End BLOCK 3 and begin BLOCK 4 */
/* Step successful. Correct predicted solution, evaluate derivatives
   using corrected solution and update differences. Determine best order
   and step-size for next step. */
            *kold = *k;
            *hold = *h;

/* Correct and evaluate */
            tmp = (*h)*g[*k];
            if ( nornd ) {
                for ( m=0 ; m<n ; m++ )
                        y[m] = p[m]+tmp*(yp[m]-phi[m][0]);
            } else {
                for ( m=0 ; m<n ; m++ ) {
                    tmp1 = tmp*(yp[m]-phi[m][0])-phi[m][15];
                    y[m] = p[m]+tmp1;
                    phi[m][14] = (y[m]-p[m])-tmp1;
                }
            }
            (*fcn)( *x, y, yp);

/* Update differences for next step */
            for ( m=0 ; m<n ; m++ ) {
                phi[m][*k] = yp[m]-phi[m][0];
                phi[m][kp1] = phi[m][*k]-phi[m][kp1];
            }
            for ( i=0 ; i<=km1 ; i++ ) {
                for ( m=0 ; m<n ; m++ ) phi[m][i] += phi[m][*k];
            }

/* Estimate error at order "kp1" unless either in first phase when
   always raise order or have already decided to lower order or
   step-size not constant so estimate unreliable */
```

```
            erkp1 = 0.0;
            if ( knew==km1 || *k==12 ) phase1 = 0;
            if ( phase1 ) {                 /* Raise order */
                *k = kp1;
                erk = erkp1;
            } else if ( knew==km1 ) {       /* Lower order */
                *k = km1;
                erk = erkm1;
            } else if ( *k <= ns ) {
                for ( m=0 ; m<n ; m++ )
                            erkp1 += pow(phi[m][kp1]/wt[m],2);
                erkp1 = absh*gstr[*k]*sqrt(erkp1);

/* Using estimated error at order "kp1", determine oredr for next step */
                if ( *k > 1 ) {
                    if ( erkm1<=erk && erkm1<=erkp1 ) {
                                                    /* Lower order */
                        *k = km1;
                        erk = erkm1;
                    } else if ( erkp1<erk && *k!=12 ) {
                                                    /* Raise order */
                        *k = kp1;
                        erk = erkp1;
                    } else {
                    }
                } else if ( erkp1 < 0.5*erk ) {

/* Here "erkp1 < erkp < max(erkm1,erkm2)" else order would have been
   lowered in BLOCK 2. Thus order is to be raised */
                    *k = kp1;
                    erk = erkp1;
                } else {
                }
            } else {
            }

/* With new order determine suitable step-size for next step */
            hnew = 2.0*(*h);
            if ( !phase1 && ( halfeps < erk*two[*k] ) ) {
                hnew = *h;
                if ( halfeps < erk ) {
                    tmp = 1.0/((double) *k+1);
                    hnew = pow(halfeps/erk,tmp);
                    if ( hnew > 0.9 ) hnew = 0.9;
                    if ( hnew < 0.5 ) hnew = 0.5;
                    hnew *= absh;
                    hmin = four_rnderr*fabs(*x);
                    if ( hnew < hmin ) hnew = hmin;
                    hnew = ( *h >= 0.0 ) ? hnew : -hnew;
                }
            }
            *h = hnew;
            return;
        }
    }

/* End BLOCK 4 */
}
```