UNIVERSITY OF LEEDS

Efficient Numerical Population Density Techniques with an Application in Spinal Cord Modelling

Hugh Spencer Frederick Osborne

Submitted in accordance with the requirements for the degree of Doctor of Philosophy

The University of Leeds

Faculty of Engineering

School of Computing

June 2022

Intellectual Property and Publication Statements

The candidate confirms that the work submitted is his own, except where work which has formed part of jointly authored publications has been included. The contribution of the candidate and the other authors to this work has been explicitly indicated below. The candidate confirms that appropriate credit has been given within the thesis where reference has been made to the work of others.

Contributions to each article

Paper A (Chapter 2)

Osborne, H., Lai, Y.M., Lepperød, M.E., Sichau, D., Deutz, L. and de Kamps, M., 2021. MIIND: a model-agnostic simulator of neural populations. Frontiers in Neuroinformatics, p.28.

The article outlines the use of the MIIND software, the implementation of the "mesh method" on both the CPU and GPU, and the XML-style simulation file format which were developed before the start of this project by the named authors except HO. Some initial work was carried out by ML to improve the Python user interface. HO implemented the "grid method" on the CPU and GPU, expanded the Python interface to include a command line program and an interface for Python scripts, and updated the MIIND software to be more easily installed and run. HO also wrote the article under the supervision of MdK.

Paper B (Chapter 3)

Osborne, H. and de Kamps, M., 2022. A Numerical Population Density Technique for N-Dimensional Neuron Models (Provisionally Accepted to Frontiers in Neuroinformatics)

All work in extending the grid method to N dimensions was carried out by HO, who wrote the article with supervision from MdK.

Paper C (Chapter 4)

York, G., Osborne, H., Sriya, P., Astill, S., de Kamps, M. and Chakrabarty, S., 2022. The effect of limb position on a static knee extension task can be explained with a simple spinal cord circuit model. Journal of Neurophysiology, 127(1), pp.173-187.

The original experimental design and data collection were performed by PS, SA, and SC. Initial data analysis including early NMF, VAF and correlation calculations was done by GRY. Further NMF and direct analysis of the sEMG signals was performed by HO with supervision from SC. All modelling and simulation work was carried out by HO with supervision from MdK. HO wrote the article with the exception of figure 4.4 in section 4.3.2 by GRY and section 4.2.3 by PS and SC.

This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

© 2022 The University of Leeds, Hugh Spencer Frederick Osborne

Acknowledgements

I wish to thank my supervisor, Marc de Kamps, for unwavering academic and pastoral support. The inevitable weight of looming deadlines and imposter syndrome were alleviated by his commitment to a good work-life balance and positive, constructive feedback. From the very beginning of the project, he gave me so much freedom and offered numerous research avenues to explore.

I would also like to thank my paper A co-authors, Yi Ming Lai, Mikkel Lepperød, David Sichau, and Lukas Deutz for their contributions to the MIIND software and informative discussions regarding its use and design.

Early in the project I was introduced to Samit Chakrabarty and his lab and am proud to have been able to work with them on such an interesting study. I wish to thank Samit and the group for being so welcoming and providing space for lively and informative discussions. I would particularly like to thank Gareth York for his excellent early data analysis and commitment to the work presented in paper C. The experiment in paper C was carried out by Piyanee Sriya and Sarah Astill and I thank them for an excellent data set and explanation of the protocol.

From the computing department at the University of Leeds, I would like to thank Netta Cohen and “the worm guys” for being a willing audience for presentation practice and asking such incisive questions early on in my project.

My project was funded by the EPSRC and I wish to thank them for their continued support for research in the UK. Finally, to the source of my constant joy and strength, Sarah, and our stunning daughter, Abi, thank you.

Abstract

MIIND is a neural simulator which uses an innovative numerical population density technique to simulate the behaviour of multiple interacting populations of neurons under the influence of noise. Recent efforts have produced similar techniques but they are often limited to a single neuron model or type of behaviour. Extensions to these require a great deal of further work and specialist knowledge. The technique used in MIIND overcomes this limitation by being agnostic to the underlying neuron model of each population. However, earlier versions of MIIND still required a high level of technical knowledge to set up the software and involved an often time-consuming manual pre-simulation process. It was also limited to only two-dimensional neuron models.

This thesis presents the development of an alternative population density technique, based on that already in MIIND, which reduces the pre-simulation step to an automated process. The new technique is much more flexible and has no limit on the number of time-dependent variables in the underlying neuron model. For the first time, the population density over the state space of the Hodgkin-Huxley neuron model can be observed in an efficient manner on a single PC. The technique allows simulation time to be significantly reduced by gracefully degrading the accuracy without losing important behavioural features. The MIIND software itself has also been simplified, reducing technical barriers to entry, so that it can now be run from a Python script and installed as a Python module.

With the improved usability, a model of neural populations in the spinal cord was simulated in MIIND. It showed how afferent signals can be integrated into common reflex circuits to produce observed patterns of muscle activation during an isometric knee extension task. The influence of proprioception in motor control is not fully understood as it can be both task and subject-specific. The results of this study show that afferent signals have a significant effect on sub-maximal muscle contractions even when the limb remains static. Such signals should be considered when developing methods to improve motor control in activities of daily living via therapeutic or mechanical means.

Contents

List of Figures

List of Tables

Abbreviations

AdEx/Adexp	Adaptive exponential integrate-and-fire
ANN	Artifical neural network
BF	Biceps femoris
CLI	Command line interface
CNS	Central nervous system
CPG	Central pattern generator
CPU	Central processing unit
CSR	Compressed sparse row
CUDA	Compute unified device architecture
E-I	Excitatory-Inhibitory
EIF	Exponential integrate-and-fire
EMG	Electromyogram / electromyographic
GIF	Generalised integrate-and-fire
GPU	Graphics processing unit
GPGPU	General purpose graphics processing unit
HPC	High performance computing
LIF	Leaky integrate-and-fire
MG	Medial gastrocnemius
MIIND	Multiple instantiations of interacting neural dynamics
MN	Motor neuron
MVC	Maximum voluntary contraction
ND	N-dimensional
NMF	Non-negative matrix factorisation
ODE	Ordinary differential equation
PDT	Population density technique
QIF	Quadratic integrate-and-fire
RDM	Refractory density method
RF	Rectus femoris
sEMG	Surface EMG
ST	Semitendinosus
TA	Tibialis anterior
TVB	The virtual brain
UML	Unified modelling language
VAF	Variance accounted for
VL	Vastus lateralis
VM	Vastus medialis
XML	Extensible markup language

Chapter 1

Introduction

The Hodgkin-Huxley neuron model is a stalwart of theoretical neuroscience. Based on the mechanics of ion channel gating of sodium and potassium ions, it provides a common structure for modellers to describe many neural behaviours in terms of time-dependent membrane conductance variables. In a population of such neurons, what distribution of values do the variables take in response to input and noise? What values can variables *not* take? How does population behaviour differ from that of the individual neurons and what happens to the distribution of values when multiple populations interact? Simulating individual neurons with a so-called Monte Carlo approach goes some way to answering these questions but only with the aid of summarising techniques such as averaging or binning. It also requires large numbers of neurons which is computationally expensive and verbose. Mean-field techniques for simulating neural populations rely on abstraction from the behaviour of the underlying neurons or other approximations to reduce the number of variables. This thesis demonstrates a new population density technique (PDT) that puts no limit on the number of time-dependent variables and requires no more than the definition of the underlying neuron model to simulate a population. The software that implements it, MIIND, can render the resulting distribution of values while the simulation is running on a single PC with reasonable time efficiency.

Fig. 1.1 shows the distribution of gating and membrane potential values in a population of Hodgkin-Huxley neurons under the influence of noisy Poisson distributed input. The panels show different perspectives for different combinations of three of the four time-dependent variables of the underlying neuron model. Bright, opaque volumes of state space indicate that a large proportion of neurons in the population have those values. No neurons can be found with variables in transparent volumes. The

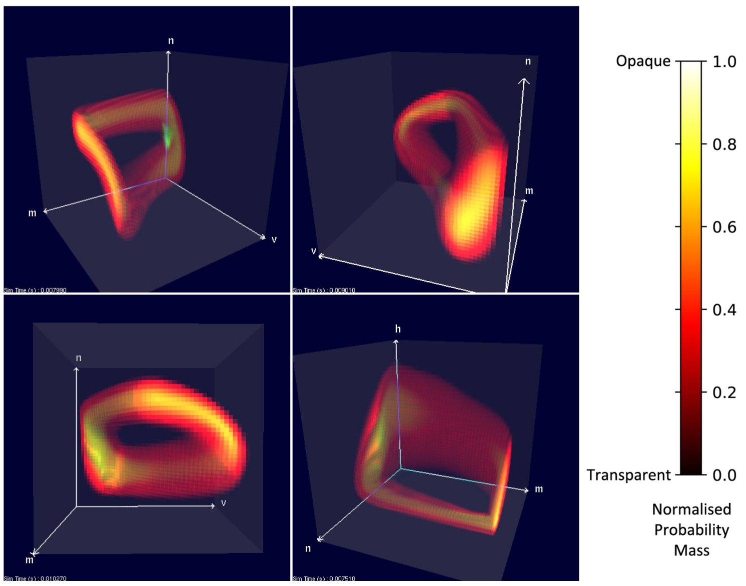


Figure 1.1: The steady state probability mass function of a population of Hodgkin-Huxley neurons rendered from four different perspectives in MIIND. Brighter, more opaque volumes indicate a higher probability mass (a higher probability of finding neurons with that state). Three perspectives show the marginal distribution for the activation variables, n and m and the membrane potential, v, from different directions. The bottom right panel shows the marginal distribution for the gating variables only.

PDT is an unusually geometric method and the rendered distribution is a direct representation of the data being processed in the algorithm. Summary metrics can be generated from the distribution such as average membrane potential or, for neuron models with a threshold-reset mechanism, the average firing rate. MIIND can simulate multiple populations which interact via these average activity metrics to reproduce complex population-level behaviours. The ability of MIIND to simulate the population in Fig. 1.1 is a culmination of work to improve the usability of the existing software, develop a technique for automating the simulation setup, and then extend that technique to higher numbers of variables/dimensions. Three standalone articles are presented in this thesis. Papers A and B describe the work done on the MIIND simulator and the PDT.

1.1 What is a population density technique?

A single simulated neuron can be described by its state, a set of time-dependent features such as membrane potential, excitability, or previous spike time. A neuron model is a parameterised definition of how the state changes in time. When simulating a population of neurons, the Monte Carlo approach, preferred by simulators such as NEST (Gewaltig and Diesmann, 2007), calculates the time evolution of the state of each individual neuron according to the model. Many neurons receive spikes from others in the population, which causes a change in the state, or they can receive a randomly distributed spike train. In contrast to the Monte Carlo approach, PDTs calculate the time-evolution of a probability density function defined over all possible states. The probability density function answers the question: what is the probability of finding a neuron from the population with a given state? Various approaches have been developed to calculate the time evolution of the density function from which population-level metrics can be derived. However, the two important processes which, for all PDTs, govern the change in the probability density function are the dynamics of the underlying neuron model (to which all neurons adhere) and the state changes caused by random incoming spikes.

Due to advances in parallelisation, particularly with the advent of general purpose graphics processing unit (GPGPU) programming, Monte Carlo or direct simulation can be performed on up to 10000 neurons in real-time on a single card (J. C. Knight et al., 2021). The speed is mostly tied to the number of neurons in the population and the complexity of the neuron model plays less of a role. For PDTs, the limiting factor is the complexity of the model alone. Early PDTs (B. W. Knight, 1972; Omurtag et al., 2000) dealt with one-dimensional integrate-and-fire neuron populations where the probability density function was defined over membrane potential. They reproduced population behaviour much quicker than direct simulation (Monte Carlo). However, techniques for density functions with two or more time-dependent variables (Apfaltrer et al., 2006; Schwalger and Lindner, 2015; de Kamps et al., 2019), show diminishing returns in this respect. Therefore, the development of most recent PDTs is concerned with maximising the repertoire of phenomenological behaviours they can produce while keeping the dimensionality of the density function as low as possible. Given that, by design, the population behaviour is dependent on the underlying neuron model, these two factors are in conflict. Relatedly, greater restrictions on which population behaviours can be approximated with a given technique reduce the chances for it to be widely accepted as a simulation method.

Simulation speed is not the only factor when considering the benefits of PDTs. The probability density function is, in a sense, the idealised representation of a neural population which can only be estimated from a direct simulation. For example, to find the steady state firing rate of a population of individual neurons simulated with the Monte Carlo approach, the number of spikes must be summed and averaged over a reasonably chosen time window to account for variation due to noise. In a numerical PDT, no such variation exists (unless desired in some techniques) and a single value can be read off. Analytically solvable PDTs can even give the firing rate in terms of the model parameters without the need to simulate (Johannesma, 1969). This is important not only for theoretical analysis of a population model, but also when making comparisons with experimental data. Why should a researcher accept uncertainty in both experimental and simulated results, even if the variance in the simulated results is minimised or accounted for? In cases where the individual realisations of neuron behaviour are not relevant, PDTs give a clearer answer to what should be expected from experimental recordings.

1.2 The MIIND Population Density Technique in Brief

The new PDT developed in this project has been dubbed the "grid method" and is an alternative form of the original MIIND PDT "mesh method" proposed by de Kamps, 2003. Papers A and B describe in detail the implementation of both methods. However to summarise, in both methods, the probability density function is defined across the time-dependent variables of the underlying neuron model as it is in the work by Omurtag et al., 2000 and others (Johannesma, 1969; B. W. Knight, 1972). This is in contrast to other more recent PDTs (Chizhov et al., 2006) such as refractory density methods (RDMs) which define the probability density function over neuron age (time since the last spike). The state space of the underlying neuron model in MIIND is discretised so that finite volumes of state space (cells) are associated with a probability mass value which is assumed to be uniformly distributed across each cell. In the mesh method, the discretisation is a set of quadrilateral cells built from the characteristic curves of the underlying neuron model. In the grid method, the discretisation is a regular set of rectangles which later are extended to three-dimensional cuboids and beyond. When probability mass moves from one cell to another, this is an indication that neurons in the population have changed state. The state change of the neurons in the population, and thus the change to probability mass, is caused by the two separate processes mentioned earlier: the deterministic time evolution of the neuron model, and the random arrival of spikes from other neurons in other populations which in MIIND are generally considered to be Poisson distributed. The MIIND simulation begins with an initial probability density function and iterates forward in time, solving the effect of both processes on the discrete cells. When the GPGPU implementation of MIIND is run on the graphics card of a single PC, even the four-dimensional Hodgkin-Huxley population can be simulated in a reasonable time. Paper B shows that the three-dimensional population simulations in MIIND even approach real time on only a moderately powerful machine.

1.3 “MIIND is Quirky!”

Particularly since the development of the two-dimensional mesh method (de Kamps et al., 2019), the probability density heat plots generated by MIIND draw many a conference-goer’s eye. Directly observing the spread of probability mass in a synaptic conductance variable due to incoming noise, or the jump in the adaptation variable after spikes in a population of adaptive exponential integrate-and-fire neurons, is a unique aspect of the simulator and the PDT. While the behaviour of a population is intuitive at first glance, the distribution of probability mass can go to unexpected areas of state space. For example, in a population of Fitzhugh-Nagumo neurons with no overall incoming excitatory drive but with enough noise, small numbers of neurons from the population can reach threshold and produce an action potential. MIIND shows this in Fig. 1.2 as a light band of probability mass around the limit cycle.

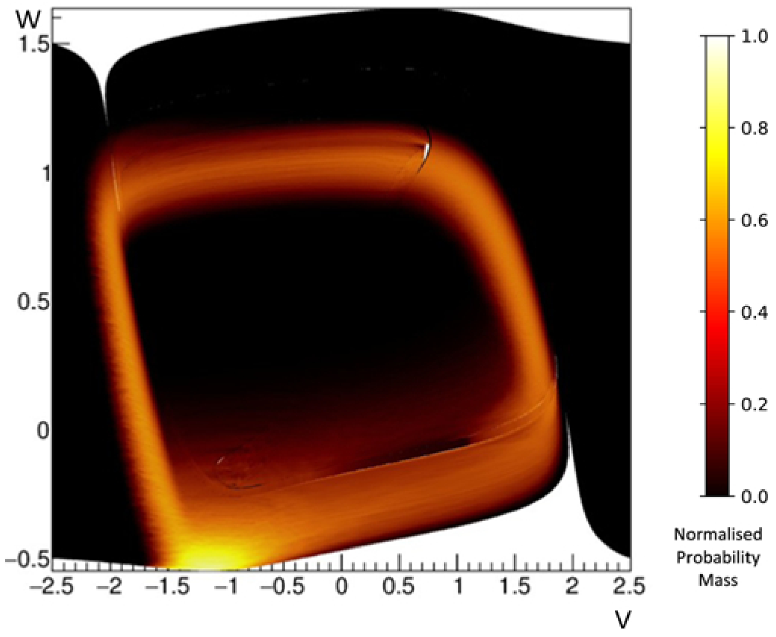


Figure 1.2: The Fitzhugh-Nagumo neuron model state space is made up of the membrane potential, V, and recovery variable, W, in arbitrary units. With no overall excitatory input, the majority of the mass in the probability function remains near the resting potential at -1.2 shown by the bright yellow area. Due to noise, however, some mass escapes and is carried round the limit cycle, indicating that there is a small possibility for neurons to fire action potentials.

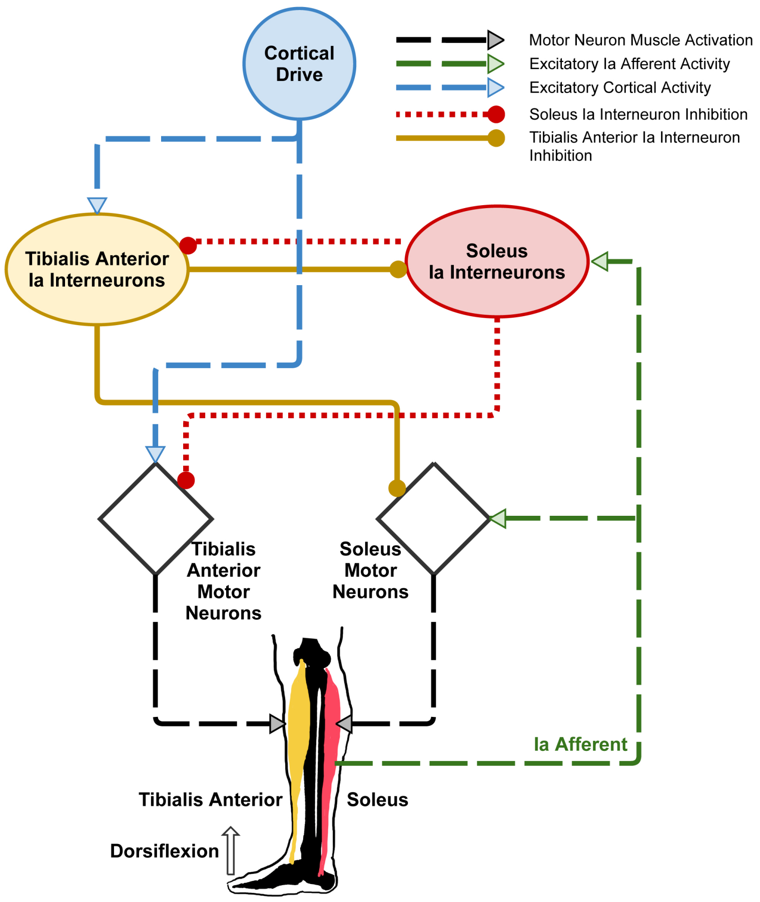
In paper B, being able to render the probability density function makes it immediately clear what causes the interesting oscillatory patterns in the average membrane potential of neurons in an E-I population network. The ability to render the probability density function so easily in MIIND comes from the geometric nature of the PDT itself. In contrast to many PDTs, both the grid and mesh methods can be described without the need for significant theoretical understanding. In all the articles presented here, the mathematical underpinning of the methods is only minimally referenced in favour of the technical implementation. An intuitive understanding of the technique is important to increase the confidence of researchers using the software and to improve understanding among those interpreting their results. That MIIND has now been used in a modelling study of the spinal cord is a testament to this.

1.4 Investigating Afferent Signals in Spinal Cord Circuits

Paper C describes a novel experimental protocol for investigating muscle activity during an isometric task, based on similar clinical tests to measure muscle stretch and limb mobility in patients with motor impairments. The motivation for devising such an experiment is that more traditional protocols are far more tightly controlled than the tasks of everyday living or simple clinical tests. While this is useful for identifying specific behaviours among single or pairs of muscles, in reality, a multitude of muscles is recruited for even simple tasks. Furthermore, due to resource practicalities, many clinical tests lack the specialised equipment available in a lab. Designing experiments with reduced or widely available equipment based on more natural movements may make it easier to match motor control behaviours observed in experiments to those in a clinical or everyday setting. The downside of this approach, however, is that care must be taken to identify features of the studied task which are truly held constant and those which are not. Initially it appeared as though the experiment was a purely isometric maximal voluntary contraction task; this is a common protocol where, with maximal effort, a subject attempts to perform a movement with a limb that is held in place. The knee was held by a brace, subjects appeared to be maximally activating their muscles, and the leg stayed still. However, because the subjects were able to freely flex their hip, maximal contractions of the knee extensor muscles should have caused the leg to lift. Realising that subjects were deliberately co-contracting their muscles to hold the leg down required both a biomechanical and even psychological understanding of the experiment.

Once the mechanics of the experiment were better understood, a prediction could be made about what kinds of afferent signals might be generated by the muscles and how changing the leg position might affect them. At this point, a neural population model was required to understand how the changes in afferent signal might lead to the observed changes in muscle activity. With the understanding that the recorded muscles included both extensors and flexors of the knee, an initial prototype model was set up in MIIND based on the well-known Ia antagonist inhibition circuit with monosynaptic excitation of the homonymous (the same) muscle. In response to the stretching of a muscle, the reflex causes additional excitation on that muscle and additional inhibition on the antagonist. MIIND produced clear and reproducible firing rate activity from the neural populations. This eliminated any uncertainty in the simulated results for a better comparison with the sEMG time series. However, this first attempt produced the reverse trend to what was observed in the experiment. Luckily, MIIND allows prototype models to be built quickly and conveniently using the XML-style simulation file and tested from a Python script for further analysis of the output. Another known reflex circuit was tried, autogenic inhibition, that in response to an increased force on a muscle, inhibits the homonymous (the same) muscle and in some circumstances, excites the antagonist muscles. This was a far better fit given the estimated trends in afferent activity. Additional populations and inputs were added to account for further significant results from the experimental data. Having run previous network models in MIIND and directly observed the behaviour of populations via the probability density function before the prototypes were built, there was already a better understanding of how populations might interact in a given model. For example, the two mutually inhibiting populations of spinal interneurons (and also shown in Fig. 1.3 below) might have caused oscillatory behaviour depending on the transmission delay of inhibitory signals between them, or the dynamics of the underlying neuron models. With no oscillations visible in the sEMG recordings, this immediately ruled out the likelihood of interneurons requiring a model containing a slowly changing excitability variable such as a burster or an adaptation variable with a fast time scale. Though the final model presented in paper C cannot answer all questions about the observed results, useful context was added to the experimental observations and certain afferent mechanisms could be ruled out.

The function of neural mechanisms such as the one described by the model is crucial to understand many common diseases that diminish motor control. In particular, many neurological diseases or spinal cord injuries result in the appearance of spasticity (stiffness during movement due to the increased excitability of motor neurons). Even when the pathology occurs supraspinally, for instance in stroke patients, spinal interneuron circuits are implicated in these symptoms. For example, the Ia reciprocal inhibition of antagonistic muscles increases inhibition on soleus motor neurons and reduces inhibition on those of tibialis anterior during dorsiflexion of the ankle. Reduced cortical signals to the Ia inhibitory interneurons due to stroke or multiple sclerosis (as shown in Fig. 1.3) can reverse this reciprocity with the effect of reducing the ability of tibialis anterior to lift the foot (Crone et al., 1994). The involvement of afferent pathways in these neural circuits is important because it potentially provides an avenue for influencing the activity of spinal circuits, and therefore motor function, without the danger of invasive procedures.



Cortical Drive
Motor Neuron Muscle Activation
Excitatory Ia Afferent Activity
Excitatory Cortical Activity
Soleus Ia Interneuron Inhibition
Tibialis Anterior Ia Interneuron Inhibition
Tibialis Anterior Ia Interneurons
Soleus Ia Interneurons
Tibialis Anterior Motor Neurons
Soleus Motor Neurons
Ia Afferent
Tibialis Anterior
Soleus
Dorsiflexion

Figure 1.3: A neural population circuit showing a mechanism for spasticity in soleus involving the Ia reflex circuit. Reduced cortical drive, perhaps due to multiple sclerosis, to the tibialis anterior Ia interneuron population, reduces the inhibition of the soleus motor neuron and soleus Ia interneuron populations which increases the inhibition of the tibialis anterior motor neurons. The soleus is more active and tibialis anterior is less active which limits the ability to lift the foot.

Wearable devices or therapies developed for those with movement disorders have used electrical stimulation to directly activate muscles or passive mechanical restrictions (braces) to improve gait or active tasks (Schuhfried et al., 2012). From the earliest studies, it has been known that vibration can be used to stimulate sensory afferents (Matthews, 1969; Hultborn et al., 1987) and recently, a glove device was developed to improve finger sensation in post-stroke patients (Bolognini et al., 2016). Beyond direct intervention from wearable devices, a better understanding of neural mechanisms involving afferent feedback could improve outcomes from current physiotherapy techniques.

Though the content of papers A and B differs from that of paper C, taken together, there is a path for future applications of population density techniques outside of cortex. The extension of MIIND to N dimensions means that models can be developed to match many more population-level behaviours from microscopic features even beyond neural dynamics.

Chapter 2

MIIND : A Model-Agnostic Simulator of Neural Populations (Paper A)

2.1 Abstract

MIIND is a software platform for easily and efficiently simulating the behaviour of interacting populations of point neurons governed by any 1D or 2D dynamical system. The simulator is entirely agnostic to the underlying neuron model of each population and provides an intuitive method for controlling the amount of noise which can significantly affect the overall behaviour. A network of populations can be set up quickly and easily using MIIND's XML-style simulation file format describing simulation parameters such as how populations interact, transmission delays, post-synaptic potentials, and what output to record. During simulation, a visual display of each population's state is provided for immediate feedback of the behaviour and population activity can be output to a file or passed to a Python script for further processing. The Python support also means that MIIND can be integrated into other software such as The Virtual Brain. MIIND's population density technique is a geometric and visual method for describing the activity of each neuron population which encourages a deep consideration of the dynamics of the neuron model and provides insight into how the behaviour of each population is affected by the behaviour of its neighbours in the network. For 1D neuron models, MIIND performs far better than direct simulation solutions for large populations. For 2D models, performance comparison is more nuanced but the population density approach still confers certain advantages over direct simulation. MIIND can be used to build neural systems that bridge the scales between an individual neuron model and a population network. This allows researchers to maintain a

plausible path back from mesoscopic to microscopic scales while minimising the complexity of managing large numbers of interconnected neurons. In this paper, we introduce the MIIND system, its usage, and provide implementation details where appropriate.

2.2 Introduction

2.2.1 Population-Level Modelling

Structures in the brain at various scales can be approximated by simple neural population networks based on commonly observed neural connections. There are a great number of techniques to simulate the behaviour of neural populations with varying degrees of granularity and computational efficiency. At the highest detail, individual neurons can be modelled with multiple compartments, transport mechanisms, and other biophysical attributes. Simulators such as GENESIS (M. A. Wilson et al., 1988; Bower and Beeman, 2012) and NEURON (Hines and Carnevale, 2001) have been used for investigations of the cerebellar microcircuit (D'Angelo et al., 2016) and a thalamocortical network model (Traub et al., 2005). Techniques which simulate the individual behaviour of point neurons such as in NEST (Gewaltig and Diesmann, 2007), or the neuromorphic system SpiNNaker (Furber et al., 2014), allow neurons to be individually parameterised and connections to be heterogeneous. This is particularly useful for analysing information transfer such as edge detection in the visual cortex. They can also be used to analyse so called finite-size effects where population behaviour only occurs as a result of a specific realisation of individual neuron behaviour. There are, however, performance limitations on very large populations in terms of both computation speed and memory requirements for storing the spike history of each neuron.

At a less granular level, rate-based techniques are a widely used practice of modelling neural activity with a single variable, whose evolution is often described by first-order ordinary differential equations, which goes back to H. Wilson and Cowan, 1972. The Virtual Brain (TVB) uses these types of models to represent activity of large regions (nodes) in whole-brain networks to generate efficient simulations (Sanz Leon et al., 2013; Jirsa et al., 2014). TVB demonstrates the benefits of a rate-based approach with the Epileptor neural population model yielding impressive clinical results (Proix et al., 2017). The Epileptor model is based on the well-known Hindmarsh-Rose neuron model (Hindmarsh and Rose, 1984). However, the behaviour of this and other rate-based models is defined at the population level instead of behaviour emerging from a definition of the underlying neurons. Therefore, these models have less power to explain simulated behaviours at the microscopic level.

Between these two extremes of granularity is a research area which bridges the scales by deriving population-level behaviour from the behaviour of the underlying neurons. So-called population density techniques (PDTs) have been used for many years (Knight, 1972; Knight et al., 1996; Omurtag et al., 2000) to describe a population of neurons in terms of a probability density function. The transfer function of a neuron model or even an experimental neural recording can be used to approximate the response from a population using this technique (H. Wilson and Cowan, 1972; El Boustani and Destexhe, 2009; Carlu et al., 2020). However, analytical solutions are often limited to regular spiking behaviour with constant or slowly changing input. The software we present here, MIIND, provides a numerical solution for populations of neurons with potentially complex behaviours (for example bursting) receiving rapidly changing noisy input with arbitrary jump sizes. The noise is usually assumed to be shot noise, but MIIND can also be used with other renewal processes such as Gamma distributed input (Lai and de Kamps, 2017). It contains a number of features that make it particularly suitable for dynamical systems representing neuronal dynamics, such as an adequate handling of boundary conditions that emerge from the presence of thresholds and reset mechanisms but is not restricted to neural systems. The dynamical systems can be grouped in large networks, which can be seen as the model of a neural circuit at the population level.

The key idea behind MIIND is shown in Fig. 2.1A. Here, a population of neurons is simulated. In this case, the neurons are defined by a conductance-based leaky-integrate-and-fire neuron model with membrane potential and state of the conductance as the two variables. The neuron's evolution through state space is given by a two-dimensional dynamical system described by Eq. 2.1.

$$\begin{aligned} \tau \frac{dV}{dt} &= -g_l(V - E_l) - g_e(t)V, \\ \tau_e \frac{dg_e}{dt} &= -g_e + I_{syn}(t) \end{aligned} \tag{2.1}$$

V is the membrane potential and g_e is the conductance variable. E_l (set to -65mV in this example) is the reversal potential and τ (20ms) and τ_e (5ms) represent the time scales for V and g_e respectively. I_{syn} represents changes to the conductance variable due to incoming spikes. If V is raised above a specified threshold value (-55mV), it is reset to a specified reset membrane potential (-65mV).

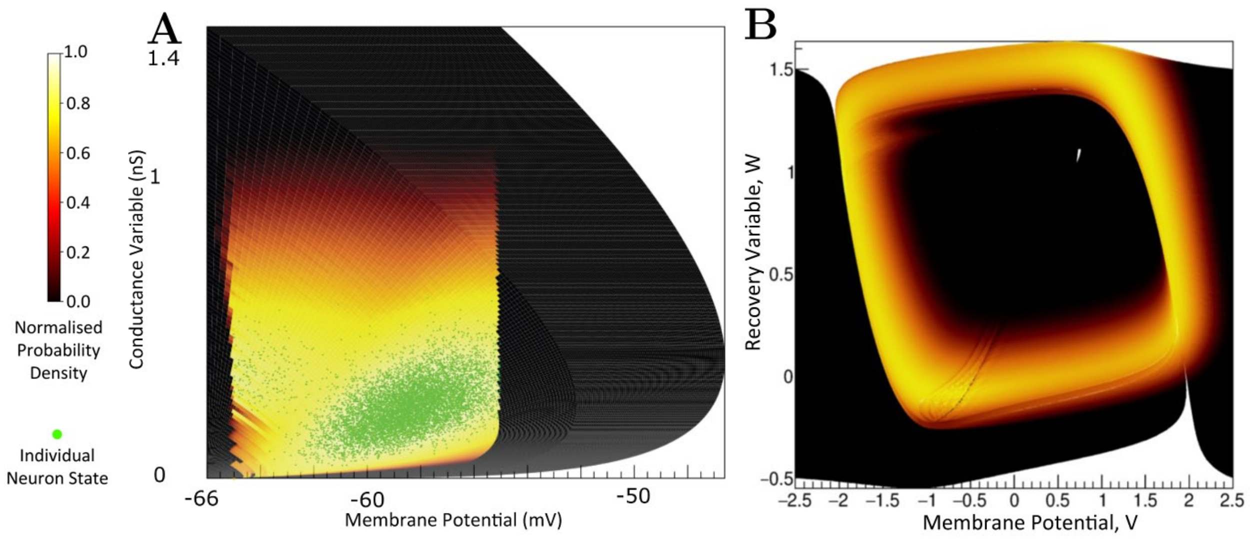


Figure 2.1: (A) The state space of a conductance-based point model neuron. It is spanned by two variables: the membrane potential and a variable representing how open the channel is. This channel has an equilibrium potential that is positive. The green dots represent the state of individual neurons in a population. They are the result of the direct simulation of a group of neurons. MIIND, however, produces the heat plot representing a density (normalised to the maximum density value) which predicts where neurons in the population are likely to be: most likely in the white areas, least likely in the red areas and not at all in the black areas. The sharp vertical cut of the coloured area at -55mV represents the threshold at which neurons are removed from state space. They are subsequently inserted at the reset potential, at their original conductance state value. (B) The state-space of a Fitzhugh-Nagumo neuron model. The axes have arbitrary units for variables V and W. There is no threshold-reset mechanism and the density follows a limit cycle. After a certain amount of simulation time, neurons can be found at all points along the limit cycle as shown here by a consistently high brightness.

The positions of individual neurons change in state space, both under the influence of the neuron's endogenous dynamics as determined by the dynamical system and of spike trains arriving from neurons in other populations, which cause rapid transitions in state space that are modelled as instantaneous jumps. For the simulation techniques mentioned earlier involving a large number of individual model neuron instances, a practice that we will refer to as Monte Carlo simulation, the population can be represented as a cloud of points in state space. The approach in MIIND, known as a population density technique, models the probability density of the cloud (shown in Fig. 2.1 as a heat map) rather than the behaviour of individual neurons. The threshold and reset values of the underlying neuron model are visible in the hard vertical edges of the density in Fig. 2.1A. In Fig. 2.1B, the same simulation approach is used for a population of Fitzhugh-Nagumo neurons (FitzHugh, 1961; Nagumo et al., 1962). The dynamical system is defined in Eq. 2.2.

$$
\begin{aligned}
\frac{dV}{dt} &= V - \frac{V^3}{3} - W, \\
\frac{dW}{dt} &= 0.08(V + 0.7 - 0.8W)
\end{aligned}
\tag{2.2}
$$

V represents the membrane potential and W is a recovery variable. The Fitzhugh-Nagumo model has no threshold-reset mechanism and so there are no vertical boundaries to the density. As well as the density function being informative in itself, common population metrics such as average firing rate and average membrane potential can be quickly derived. The MIIND model archive, available in the code repository, contains example simulation files for populations of both conductance-based neurons and Fitzhugh-Nagumo neurons (*examples/model_archive/Conductance2D* and *examples/-model_archive/FitzhughNagumo*).

2.2.2 The Case for Population Density Techniques

Why use this technique? Omurtag et al., 2000; Nykamp and Tranchina, 2000; de Kamps, 2003; Iyer et al., 2013 have demonstrated that PDTs are much faster than Monte Carlo simulation for 1D models; de Kamps, Lepperød, et al., 2019 have shown that while speed is comparable between 2D models and Monte Carlo, memory usage is orders of magnitude lower because no spikes need to be buffered, which accounts for significant memory use in large-scale simulations. In practice, this may make the difference between running a simulation on an HPC cluster or a single PC equipped with a general-purpose graphics processing unit (GPGPU).

Apart from simulation speed, PDTs have been important in understanding population-level behaviour analytically. Important questions, such as ‘why are cortical networks stable?’ (Amit and Brunel, 1997), ‘how can a population be oscillatory when its constituent neurons fire sporadically?’ (Brunel and Hakim, 1999), ‘how does spike shape influence the transmission spectrum of a population?’(Fourcaud-Trocmé et al., 2003) have been analysed in the context of population density techniques, providing insights that cannot be obtained from merely running simulations. A particularly important question, which has not been answered in full is: ‘how do rate-based equations emerge from populations of spiking neurons and when is their use appropriate?’. There are many situations where such rate-based equations are appropriate, but some where they are not and their correspondence to the underlying spiking neural dynamics is not always clear (Montbrió et al., 2015; de Kamps, 2013). There is a body of work suggesting that some rate-based equations can be seen as the lowest order of perturbations of a stationary state, and much of this work is PDT-based (H. Wilson and Cowan, 1972; Gerstner, 1998; Mattia and Del Giudice, 2002; Mattia and Del Giudice, 2004; Montbrió et al., 2015). MIIND opens the possibility to incorporate these theoretical insights into large-scale network models. For example, we can demonstrate the prediction from Brunel and Hakim, 1999 that inhibitory feedback on a population can cause a bifurcation and produce resonance. Finally, for a steady-state input, the firing rate prediction of a PDT model converges to a transfer function which can be used in artificial spiking neural networks (de Kamps, Baier, et al., 2008).

2.2.3 Population-level Modelling

For the population density approach we take with MIIND, the time evolution of the probability density function is described by a partial integro-differential equation. We give it here to highlight some of its features, but for an in-depth introduction to the formalism and a derivation of the central equations we refer to Omurtag et al., 2000.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial \vec{v}} \cdot (\frac{\vec{F}(\vec{v})\rho(\vec{v},t)}{\tau}) = \int_M d\vec{v}' \left\{ W(\vec{v} \mid \vec{v}')\rho(\vec{v'}) - W(\vec{v}' \mid \vec{v})\rho(\vec{v}) \right\}, \tag{2.3}$$

ρ is the probability density function defined over a volume of state space, M, in terms of time, t, and time-dependent variables, $\vec{v}$, under the assumption that the neuronal dynamics of a point model neuron is given by:

$$\tau \frac{d\vec{v}}{dt} = \vec{F}(\vec{v}), \tag{2.4}$$

where τ is the neuron's membrane time constant. Simple models are one-dimensional (1D). For the leaky-integrate-and-fire (LIF) neuron:

$$F(v) = -v, \tag{2.5}$$

For a quadratic-integrate-and-fire (QIF) neuron:

$$F(v) = v^2 + I, \tag{2.6}$$

where v is the membrane potential, and I can be interpreted as a bifurcation parameter. More complex models require a higher dimensional state space. Since such a space is hard to visualise and understand, considerable effort has been invested in the creation of effective models. In particular two-dimensional (2D) models are considered to be a compromise that allows considerably more biological realism than LIF or QIF neurons, but which remain amenable to visualisation and analysis, and can often be interpreted geometrically (Izhikevich, 2007). Examples are the Izhikevich simple neuron (Izhikevich, 2003), the Fitzhugh-Nagumo neuron (FitzHugh, 1961; Nagumo et al., 1962), and the adaptive exponential integrate-and-fire neuron (Brette and Gerstner, 2005), incorporating phenomena such as bursting, bifurcations, adaptation, and others that cannot be accounted for in a one-dimensional model.

$W(v \mid v')$ in Eq. 2.3 represents a transition probability rate function. The right hand side of Eq. 2.3 makes it a Master equation. Any Markovian process can be represented by a suitable choice of W. For example, for shot noise, we have

$$W(v' \mid v) = \nu(\delta(v' - v - h) - \delta(v - v')), \tag{2.7}$$

where ν is the rate of the Poisson process generating spike events. The delta functions reflect that an incoming spike causes a rapid change in state space, modelled as an instantaneous jump, h. It depends on the particular neural model in what variable the jumps take place. Often models use a so-called delta synapse, such that the jump is in membrane potential. In conductance-based models, the incoming spike causes a jump in the conductance variable (Fig. 2.1A), and the influence of the incoming spike on the potential is then indirect, given by the dynamical system's response to the sudden change in the conductance state.

MIIND produces a numerical solution to Eq. 2.3 for arbitrary 1D or 2D versions of $\vec{F}(\vec{v})$ (support for 3D versions is in development), under a broad variety of noise processes. Indeed, the right-hand side of Eq. 2.3 can be generalised to other renewal processes which cannot simply be formulated in terms of a transition probability rate function W. It is possible to introduce a right-hand side that entails an integration over a past history of the density using a kernel whose shape is determined by a Gamma distribution or other renewal process (Lai and de Kamps, 2017).

2.2.4 Quick Start Guide

Before describing the implementation details of MIIND, this section demonstrates how to quickly set up a simulation for a simple E-I network of populations of conductance-based neurons using the MIIND Python library. A rudimentary level of Python experience is needed to run the simulation. In most cases, MIIND can be installed via Python pip. Detailed installation instructions can be found via the *README.md* file of the MIIND repository and in the MIIND documentation (Osborne and de Kamps, 2021). For this example, we will use a pre-written script, *generateCondFiles.py*, to generate the required simulation files which can be found in the *examples/quick_start* directory of the MIIND repository or can be loaded into a working directory using the following python command.

```
$ python -m miind.loadExamples
```

In the *examples/quick_start* directory, the *generateCondFiles.py* script generates the simulation files, *cond.model* and *cond.tmat*.

```
$ python generateCondFiles.py
```

The contents of *generateCondFiles.py* is given in Listing 2.1. The two important parts of the script are the neuron model function, in this case named *cond()*, and the call to the MIIND function *grid_generate.generate()* which takes a number of parameters which are discussed in detail later.

Listing 2.1: *generateCondFiles.py*

```
import miind.grid_generate as grid_generate

def cond(y,t):
    E_r = -65e-3
    tau_m = 20e-3
    tau_s = 5e-3

    v = y[0];
    h = y[1];

    v_prime = ( -(v - E_r) - (h * v) ) / tau_m
    h_prime = -h / tau_s

    return [v_prime, h_prime]

grid_generate.generate(
    func = cond,
    timestep = 1e-04,
    timescale = 1,
    tolerance = 1e-6,
    basename = 'cond',
    threshold_v = -55.0e-3,
    reset_v = -65e-3,
    reset_shift_h = 0.0,
    grid_v_min = -72.0e-3,
    grid_v_max = -54.0e-3,
    grid_h_min = -1.0,
    grid_h_max = 2.0,
    grid_v_res = 200,
    grid_h_res = 200,
    efficacy_orientation = 'h')
```

The *cond()* function should be familiar to those who have used Python numerical integration frameworks such as *scipy.integrate*. It takes the two time-dependent variables defined by $y[0]$ and $y[1]$ and a placeholder parameter, t, for performing a numerical integration. In the function, the user may define how the derivatives of each variable are to be calculated. The *generate()* function requires a suitable time step, values for a threshold and reset if needed, and a description of the extent of the state space to be simulated. With this structure, the user may define any two-dimensional neuron model. The generated files are then referenced in a second file which describes a network of populations to be simulated. Listing 2.2 shows the contents of *cond.xml* describing an E-I network which uses the generated files from *generateCondFiles.py*.

Listing 2.2: *cond.xml*

```
<Simulation>
     <WeightType>CustomConnectionParameters</WeightType>
     <Algorithms>
          <Algorithm type="GridAlgorithm" name="COND" modelfile="cond.model"
   ↪ tau_refractive="0.0" transformfile="cond_0_0_0_0_.tmat" start_v="-0.065"
   ↪ start_w="0.0" >
               <TimeStep>1e-04</TimeStep>
          </Algorithm>
        <Algorithm type="RateFunctor" name="ExcitatoryInput">
            <expression>800.</expression>
        </Algorithm>
     </Algorithms>
     <Nodes>
        <Node algorithm="ExcitatoryInput" name="INPUT_E" type="EXCITATORY_DIRECT" />
        <Node algorithm="ExcitatoryInput" name="INPUT_I" type="EXCITATORY_DIRECT" />
          <Node algorithm="COND" name="E" type="EXCITATORY_DIRECT" />
          <Node algorithm="COND" name="I" type="INHIBITORY_DIRECT" />
     </Nodes>
     <Connections>
        <Connection In="INPUT_E" Out="E" num_connections="1" efficacy="0.1" delay="
   ↪ 0.0"/>
        <Connection In="INPUT_I" Out="I" num_connections="1" efficacy="0.1" delay="
   ↪ 0.0"/>
          <Connection In="E" Out="I" num_connections="1" efficacy="0.1" delay="0.001"
   ↪ />
          <Connection In="E" Out="E" num_connections="1" efficacy="0.1" delay="0.001"
   ↪ />
```

```
          <Connection In="I" Out="E" num_connections="1" efficacy="-0.1" delay="0.001
    ↪ "/>
          <Connection In="I" Out="I" num_connections="1" efficacy="-0.1" delay="0.001
    ↪ "/>
     </Connections>
     <Reporting>
          <Display node="E" />
          <Display node="I" />
          <Rate node="E" t_interval="0.001" />
          <Rate node="I" t_interval="0.001" />
     </Reporting>
     <SimulationRunParameter>
          <SimulationName>EINetwork</SimulationName>
          <t_end>0.2</t_end>
          <t_step>1e-04</t_step>
          <name_log>einetwork.log</name_log>
     </SimulationRunParameter>
</Simulation>
```

The full syntax documentation for MIIND XML files is given in section 2.5. Though more compact or flexible formats are available, XML was chosen as a formatting style due to its ubiquity ensuring the majority of users will already be familiar with the syntax. The *Algorithms* section is used to declare specific simulation methods for one or more populations in the network. In this case, a GridAlgorithm named *COND* is set up which references the *cond.model* and *cond.tmat* files. A RateFunctor algorithm produces a constant firing rate. In the *Nodes* section, two instances of *COND* are created: one for the excitatory and inhibitory populations respectively. Two *ExcitatoryInput* nodes are also defined. The *Connections* section allows us to connect the input nodes to the two conductance populations. The populations are connected to each other and to themselves with a 1ms transmission delay. The remaining sections are used to define how the output of the simulation is to be recorded and to provide important simulation parameters such as the simulation time. By running the following python command, the simulation can be run.

Listing 2.3: Run the cond.xml simulation.

```
$ python -m miind.run cond.xml
```

The probability density plots for both populations will be displayed in separate windows as the simulation progresses. The firing rate of the excitatory population can be plotted using the following commands. Fig. 2.2 shows the probability density plots for both populations and average firing rate of population E.

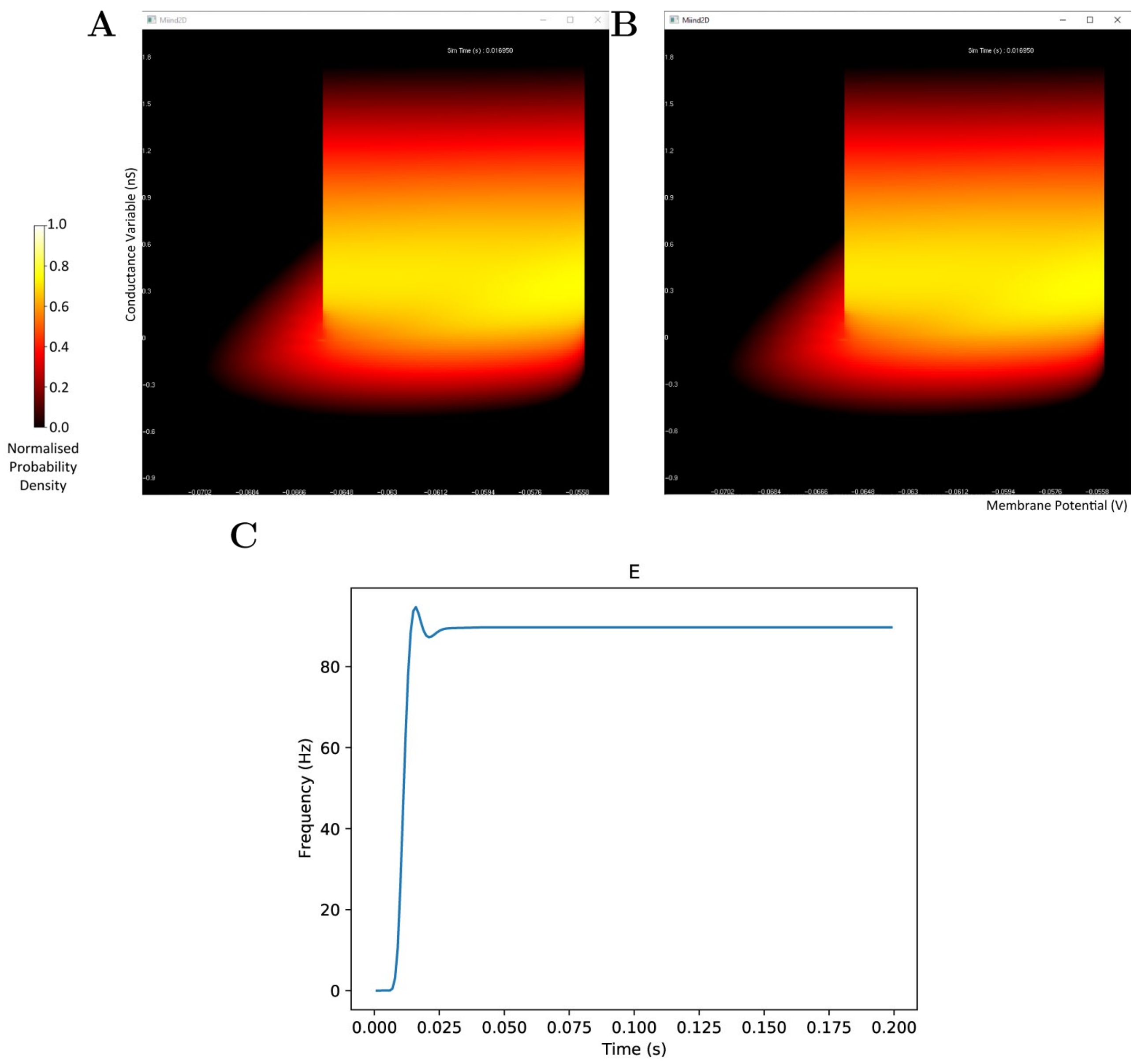


Figure 2.2: The display output of a running E-I population network simulation of conductance-based neurons. (A) The probability density heat map (normalised to the maximum density value) of the excitatory population. (B) The probability density heat map of the inhibitory population. Brighter colours indicate a larger probability mass. The axes are unlabelled in the simulation windows as the software is agnostic to the underlying model. However, the membrane potential and conductance labels have been added for clarity. (C) The average firing rate of the excitatory population.

Listing 2.4: Load the cond.xml simulation and plot the average firing rate of population E.

```
$ python -m miind.miindio sim cond.xml
$ python -m miind.miindio rate E
```

Finally, the density function of each population can be plotted as a heat map for a given time in the simulation.

Listing 2.5: Plot the probability density of population I at time 0.12s.

```
$ python -m miind.miindio plot-density I 0.12
```

Later sections will show how the MIIND simulation can be imported into a user-defined Python script so that input can be dynamically set during simulation and population activity can be captured for further processing.

2.3 The MIIND Grid Algorithm

MIIND allows the user to simulate populations of any 1D or 2D neuron model. Although much of MIIND's architecture is agnostic to the integration technique used to simulate each population, the system is primarily designed to make use of its novel population density techniques, grid algorithm and mesh algorithm. Both algorithms use a discretisation of the underlying neuron model's state space such that each discrete "cell", which covers a small area of state space, is considered to hold a uniform distribution of probability mass. In both algorithms, MIIND performs three important steps for each iteration. First, probability mass is transferred from each cell to one or more other cells according to the dynamics of the underlying neuron model in the absence of any input. The probability mass is then spread across multiple other cells due to incoming random spikes. Finally, if the underlying neuron model has a threshold-reset mechanic, such as an integrate-and-fire model, probability mass which has passed the threshold is transferred to cells along the reset potential. As it is the most practically convenient method for the user, we will first introduce the grid algorithm. We will discuss its benefits and weaknesses, indicating where it may be appropriate to use the mesh algorithm instead.

2.3.1 Generating the Grid and Transition Matrix

To discretise the state space in the grid method, the user can specify the size and $M \times N$ resolution of a rectangular grid which results in MN identical rectangular cells, each of which will hold probability mass. In the grid algorithm, a transition matrix lists the proportion of mass which moves from each cell to (usually) adjacent cells in one time step due to the deterministic dynamics of the underlying neural model. To pre-calculate the transitions for each cell, MIIND first translates the vertices of every cell by integrating each point forward by one time step according to the dynamics of the underlying neuron

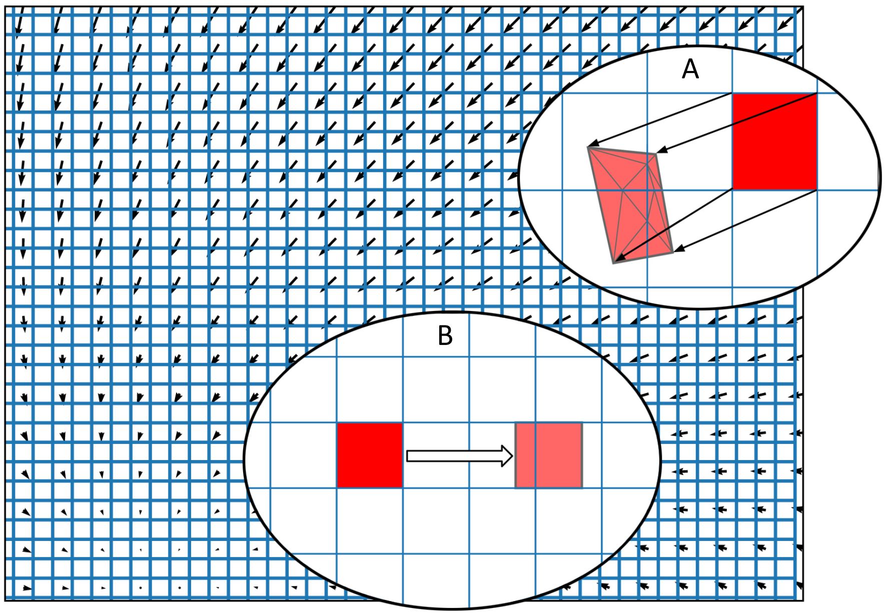


Figure 2.3: The state space of a neuron model (shown here as a vector field) is discretised into a regular grid of cells. (A) The transition matrix for solving the deterministic dynamics of the population is generated by applying a single time step of the underlying neuron model to each vertex of each cell in the grid and calculating the proportion by area to each overlapping cell. Once the vertices of a grid cell have been translated, the resulting polygon is recursively triangulated according to intersections with the original grid. Once complete, all triangles can be assigned to a cell and the area proportions can be summed. (B) For a single incoming spike (with constant efficacy), all cells are translated by the same amount and therefore have the same resulting transition which can be used to solve the Poisson master equation. In fact, the transition will always involve at most two target cells and the proportions can be calculated knowing only the grid cell width and the efficacy.

model as shown in Fig. 2.3A. As the time step is small, a single Euler step is usually all that is required to avoid large errors (although other integration schemes can be used if required). Each transformed cell is no longer guaranteed to be a rectangle and is compared to the original non-transformed grid to ascertain which cells overlap with the newly generated quadrilateral. An overlap indicates that some proportion of neurons in the original cell will move to the overlapping cell after one time step. In order to calculate the overlap, the algorithm in Listing 2.6 is employed. This algorithm is also used in the geometric method of generating transition matrices for the mesh algorithm shown later.

Listing 2.6: A pseudo-code representation of the algorithm used to calculate the overlapping areas between transformed grid cells and the original grid (or for translated cells of a mesh). The proportion of the area of the original cell gives the proportion of probability mass to be moved in each transition.

```
For each transformed cell, A, in the grid:
    Translate all four vertices according to a single Euler step.
    Split A into two triangles and add them to a triangle list.
```

```
For each non-transformed cell, B:
   Set the overlapping area sum to 0.
      While the triangle list has changed:
      For each triangle in the list:
          If the triangle is entirely outside B: add 0 to the sum.
          If the triangle is entirely within B: add the triangle's area to the sum.
          If B is entirely within the triangle: add B's area to the sum.
          Else: For each edge in B:
            Calculate any intersection points with the edges of the triangle.
            Triangulate the polygon produced by the original triangle points plus the
    ↪  new intersection points.
            Remove the original triangle from the list.
            Add the newly generated triangles to the list.
      Calculate the proportion of A taken by the sum.
      Add the transition from A to B with the proportion to the transition matrix.
```

Though the pseudo-code algorithm is order N^2, there are many ways that the efficiency of the algorithm is improved in the implementation. The number of non-transformed cells checked for overlap can be limited to only those which lie underneath each given triangle. Furthermore, the outer loop is parallelisable. Finally, as the non-transformed cells are axis-aligned rectangles, the calculation to find edge intersections is trivial. Fig. 2.3A shows a fully translated and triangulated cell at the end of the algorithm. Once the transition matrix has been generated, it is stored in a file with the extension *.tmat*. Although the regular grid can be described with only four parameters (the width, height, X, and Y resolutions), to more closely match the behaviour of mesh algorithm, the vertices of the grid are stored in a *.model* file. To simulate a population using the grid algorithm, the *.tmat* and *.model* files must be generated and referenced in the XML simulation file.

As demonstrated in the quick start guide (section 2.2.4), to generate a *.model* and *.tmat* file, the user must write a short Python script which defines the underlying neuron model and makes a call to the MIIND API to run the algorithm in listing 2.6. In the *python* directory of the MIIND source repository (see section B.1 in the supplementary material), there are a number of examples of these short scripts. The script used to generate a grid for the Izhikevich simple model is listed in the supplementary material section B.9.1. The required definition of the neuron model function is similar to those used by many numerical integration libraries. The function takes a parameter, y, which represents a list which holds the two time-dependent variables and a parameter, t, which is a placeholder for use in integration. The function must return the first time derivatives of each variable as a list in the same order as in y. Once the function has been written, a call to *grid_generate.generate* is made which takes the parameters listed in Table 2.1.

Table 2.1: Parameters for the *grid_generate.generate* function.

Parameter Name	Notes
func	The underlying neuron model function.
timestep	The desired time step for the neuron model
timescale	A scale factor for the timescale of the underlying neuron model to convert the time step into seconds.
tolerance	An error tolerance for solving a single time step of the neuron model.
basename	The base name with which all output files will be named.
threshold_v	The spike threshold value for integrate-and-fire neuron models.
reset_v	The reset value for integrate-and-fire neuron models.
reset_shift_h	A value for increasing the second variable during reset for integrate-and-fire neuron models with some adaptive shift or similar function.
grid_v_min	The minimum value for the first dimension of the grid (usually membrane potential).
grid_v_max	The maximum value for the first dimension of the grid.
grid_h_min	The minimum value for the second dimension of the grid.
grid_h_max	The maximum value for the second dimension of the grid.
grid_v_res	The number of columns in the grid.
grid_h_res	The number of rows in the grid.
efficacy_orientation	The direction, 'v' or 'h', in which incoming spikes cause an instantaneous change.

When the user runs the script, the required *.model* and *.tmat* files will be generated for use in a simulation. In the quick start guide, the conductance-based neuron model requires that *efficacy_orientation* is set to 'h' because incoming spikes cause an instantaneous change in the conductance variable instead of the membrane potential. By default, however, this parameter is set to 'v'. When choosing values for the grid bounds (*grid_v_min*, *grid_v_max*, *grid_h_min*, and *grid_h_max*), the aim is to estimate where in state-space the population density function might be non-zero during a simulation. In the conductance-based neuron model, because of the threshold-reset mechanic, the *grid_v_max* parameter need only be slightly above the threshold to ensure that there is at least one column of cells on or above threshold to allow probability mass to be reset. The *grid_v_min* value should be below the resting potential and reset potential. However, we must also consider that the neurons could receive inhibitory spikes which would cause the neurons to hyperpolarise. *grid_v_min* should therefore be set to a value beyond the lowest membrane potential expected during the simulation. Similarly, for the conductance variable, space should be provided for reasonable positive and negative values. If it is known beforehand that no inhibition will occur, however, then the state space bounds can be set tighter in order to improve the accuracy of the simulation using the same grid resolution (*grid_v_res* and *grid_h_res*). If during the simulation, probability mass is pushed beyond the lower bounds of the grid, it will be pinned at those lower bounds which will produce incorrect behaviour and results. If the probability mass is pushed beyond the upper bounds, it will be wrapped around to the lower bounds which will also produce incorrect results. The choice of grid resolution is a balance between speed of simulation and accuracy. However, even very coarse grids can produce representative firing rates and behaviours. Typical grid resolutions range between 100x100 and 500x500. It can also be beneficial to experiment with different M and N values as the accuracy of each dimension can have unbalanced influence over the population level metrics.

2.3.2 The Effect of Random Incoming Spikes

The transition matrix in the *.tmat* file describes how probability mass moves to other cells due to the deterministic dynamics of the underlying neuron model. The transition matrix is sparse as probability mass is often only transferred to nearby cells. Solving the deterministic dynamics is therefore very efficient. The mesh algorithm is even faster and, as demonstrated later, is significantly quicker than direct simulation for this part of the algorithm. Another benefit to the modeller is that by rendering the grid with each cell coloured according to its mass, the resultant heat map gives an excellent visualisation of the state of the population as a whole at each time step of the simulation as shown in Fig. 2.2. This provides particularly useful insight into the sub-threshold behaviour of neurons in the population.

The second step of the grid algorithm, which must be performed every iteration, is to solve the change in the probability density function due to random incoming spikes. It is assumed that a spike causes an instantaneous change in the state of a neuron, usually a step-wise jump in membrane potential corresponding to a constant synaptic efficacy. In the conductance-based neuron example, this jump is in the conductance. When considering each cell in the grid, a single incoming spike will cause some proportion of the probability mass to shift to at most, two other cells as shown in Fig. 2.3. Because all cells in the grid are equally distributed and the same size, the relative transition of probability mass caused by a single spike is the same for them all. A sparse transition matrix, M, can be generated from this single transition so that applying M to the probability density grid applies the transition to all cells. MIIND calculates a different M for each incoming connection to the population based on the user-defined instantaneous jump, which we refer to as the efficacy. In the mesh algorithm, the relative transitions are different for each cell and so a transition matrix (similar to that of the *.tmat* file) is required to describe the effect of a single spike. As with many other population density techniques, MIIND assumes that incoming spikes are Poisson distributed, although it is possible to approximate other distributions. MIIND uses M to calculate the change to the probability density function, ρ, due solely to the non-deterministic dynamics as described by equation 2.8.

$$d\rho/dt = \lambda M \rho \tag{2.8}$$

λ is the incoming Poisson firing rate. The boost numeric library is used to integrate $d\rho/dt$. The solution to this equation describes the spread of the probability density due to Poisson spikes. This 'master process' step amounts to multiple applications of the transition matrix M and is where the majority of time is taken computationally. However, OpenMP is available in MIIND to parallelise the matrix multiplication. If multiple cores are available, the OpenMP implementation significantly improves performance of the master process step. More information covering this technique can be found in de Kamps, Lepperød, et al., 2019; de Kamps, 2013.

2.3.3 Threshold-Reset Dynamics

Many neuron models include a "threshold-reset" process such that neurons which pass a certain membrane potential value are shifted back to a defined reset potential to approximate repolarisation during an action potential. To facilitate this in MIIND, after each iteration, probability mass in cells which lie across the threshold potential is relocated to cells which lie across the reset potential according to a pre-calculated mapping. Often, a refractory period is used to hold neurons at the reset potential before allowing them to again receive incoming spikes. In MIIND this is implemented using a queue for each threshold cell as shown in Fig. 2.4. The queues are set to the length of the refractory period divided by the time step, rounded up to the nearest integer value. During each iteration, probability mass is shifted one position along the queue. A linear interpolation of the final two places in the queue is made and this value is passed to the mapped reset cell. The interpolation is required in case the refractory period is not an integer multiple of the time step. The total probability mass in the threshold cells each iteration is used to calculate the average population firing rate. For models which do not require threshold-reset dynamics, setting the threshold value to the maximal membrane potential of the grid, and the reset to the minimal membrane potential ensures that no resetting of probability mass will occur.

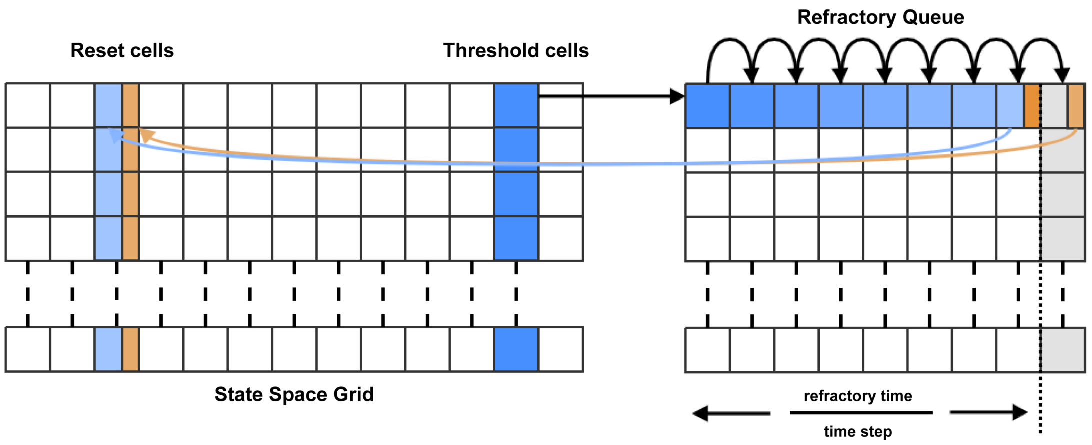


Figure 2.4: For each time step, probability mass in the cells which lie across the threshold (threshold cells) is pushed onto the beginning of the refractory queue. There is one queue per threshold cell. During each subsequent time step, the probability mass is shifted one place along the queue until it reaches the penultimate place. A proportion of the mass, calculated according to the modulo of the refractory time and the time step, is transferred to the appropriate reset cell. The remaining mass is shifted to the final place in the queue. During the next time step, that remaining mass is transferred to the reset cell.

2.3.4 How MIIND Facilitates Interacting Populations

The grid algorithm describes how the behaviour of a single population is simulated. The MIIND software platform as a whole provides a way for many populations with possibly many different integration algorithms to interact in a network. The basic process of simulating a network is as follows. The user must write an XML file which describes the whole simulation. This includes defining the population nodes of the network and how they are connected; which integration technique each population uses (grid algorithm, mesh algorithm etc.); external inputs to the network; how the activity of each population will be recorded and displayed; the length and time step of the simulation. As shown in the quick start guide, the XML file can be passed as a parameter to the *miind.run* module in Python. When the simulation is run, a population network is instantiated and the simulation loop is started. For each iteration, the output activity of each population node is recorded. By default, the activity is assumed to be an average firing rate but other options are available such as average membrane potential. The outputs are passed as inputs to each population node according to the connectivity defined in the XML file. Each population is evolved forward by one time step and the simulation loop repeats until the simulation time is up. The Python front end, *miind.miindio*, provides the user with tools to analyse the output from the simulation. A custom *run* script can also be written by the user to perform further analysis and processing.

The simplicity of the XML file means that a user can set up a large network of populations with very little effort. The model archive in the code repository holds a set of example simulations demonstrating the range of MIIND's functionality and includes an example which simulates the Potjans-Diesmann model of a cortical microcircuit (Potjans and Diesmann, 2014), which is made up of eight populations of leaky integrate-and-fire neurons. Fig. 2.5 shows a representation of the model with embedded density plots for each population.

2.3.5 Running MIIND Simulations

The quick start guide demonstrated the simplest way to run a simulation given that the required *.model*, *.tmat*, and *.XML* files have been generated. The *miind.run* script imports the *miind.miindsim* Python extension module which can also be imported into any user-written Python script. Section 2.7 details the functions which are exposed by *miind.miindsim* for use in a python script. The benefit of this method is that the outputs from populations can be recorded after each iteration and inputs can be dynamic allowing the python script to perform its own logic on the simulation based on the current state.

There is also a command line interface (CLI) program provided by the Python module, *miind.miindio*. The CLI can be used for many simple workflow tasks such as generating models and displaying results. Each command which is available in the CLI, can also be called from the MIIND Python API, upon which the CLI is built. A full list of the available commands in the CLI is given in section B.9.3 of the supplementary material and a worked example using common CLI commands is provided in section 2.8.

2.3.6 When not to use the Grid Algorithm

For many underlying neuron models, the grid algorithm will produce results showing good agreement with direct simulation to a greater or lesser extent depending on the resolution of the grid (see Fig.

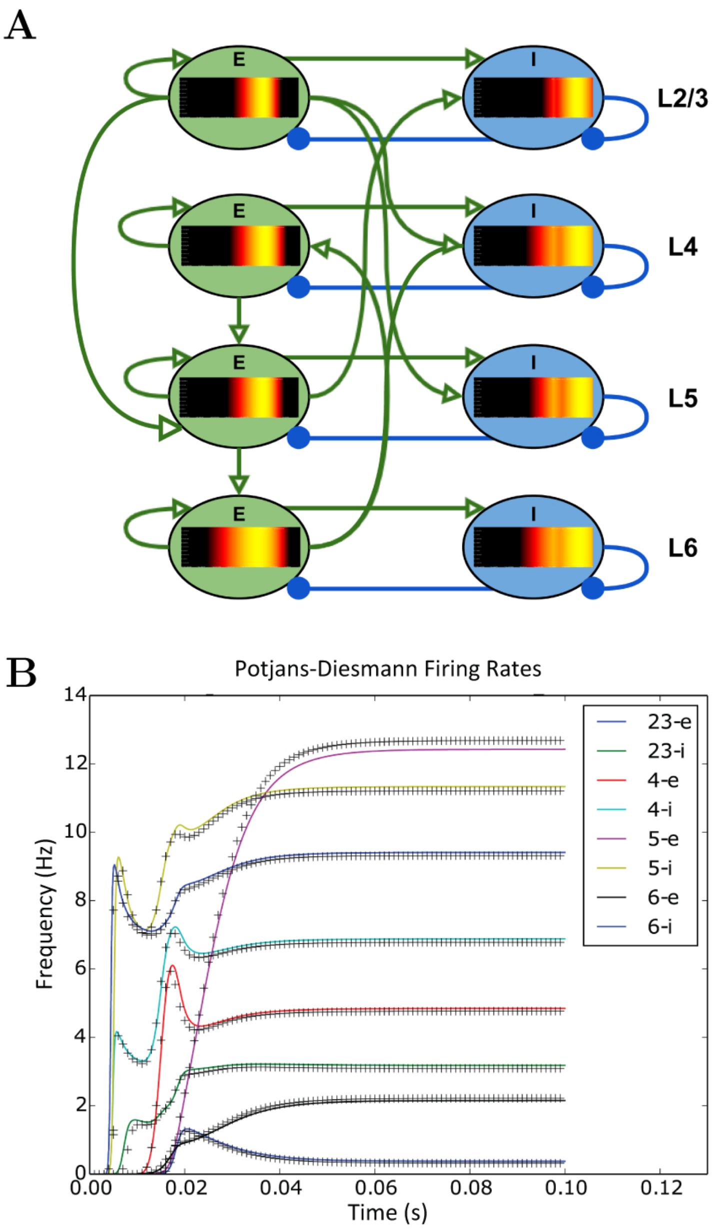


Figure 2.5: (A) A representation of the connectivity between populations in the Potjans-Diesmann microcircuit model. Each population shows the probability density at an early point in the simulation before all populations have reached a steady state. All populations are of leaky-integrate-and-fire neurons and so the density plots show membrane potential in the horizontal axis. The vertical axis has no meaning (probability mass values are the same at all points along the vertical). (B) The firing rate outputs from MIIND (crosses) in comparison to those from DiPDE for the same model (solid lines).

2.6). However, for models such as exponential integrate-and-fire, a significantly higher grid resolution is required than might be expected because of the speed of the dynamics across the threshold (beyond which, neurons perform the action potential). When the input rate is high enough to generate tonic spiking in an exponential integrate-and-fire model, the rate of depolarisation of each neuron reduces as it approaches the threshold potential then once it is beyond the threshold, quickly increases producing a spike. Because the grid discretises the state space into regular cells, if cells are large due to a low resolution, only a small number of cells will span the threshold, as shown in Fig. 2.7A. When the transition matrix is applied each time step, probability mass is distributed uniformly across each cell. Probability mass can therefore artificially cross the threshold much faster than it should, leading to a higher than expected average firing rate for the population. Using the grid algorithm for such models where the firing rate itself is dependent on sharp changes in the speed of the dynamics should be avoided if high accuracy is required. Other neuron models, like the busting Izhikevich simple model, also have sharp changes in speed when neurons transition from bursting to quiescent periods. However, the bursting firing rate is unaffected by these dynamics and the oscillation frequency is affected only negligibly due to the difference in timescales. The grid algorithm is therefore still appropriate in cases such as this. For exponential integrate-and-fire models, however, MIIND provides a second algorithm which can more accurately capture the deterministic dynamics: mesh algorithm.

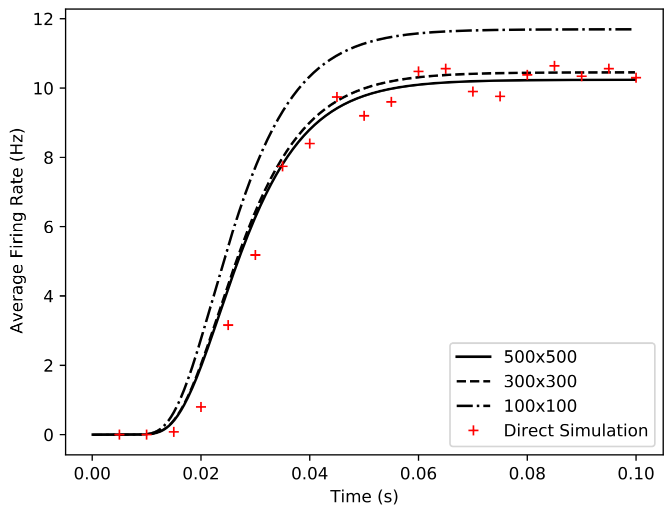


Figure 2.6: Comparison of average firing rates from four simulations of a single population of conductance-based neurons. The black solid and dashed lines indicate MIIND simulations using the grid algorithm with different grid resolutions. The red crosses show the average firing rate of a direct simulation of 10000 neurons.

2.4 The MIIND Mesh Algorithm

Instead of a regular grid to discretise the state space of the underlying neuron model, the mesh algorithm requires a two-dimensional mesh which describes the dynamics of the neuron model itself

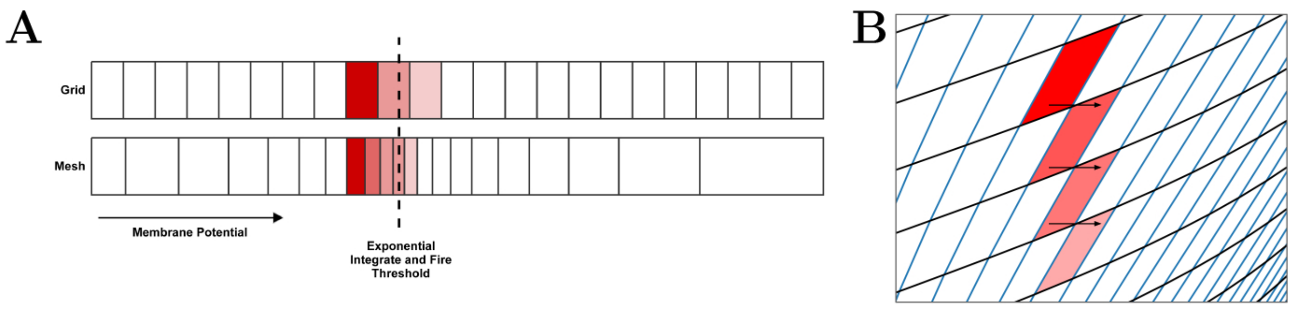


Figure 2.7: (A) In the grid algorithm, large cells cause probability mass to be distributed further than it should. This error is expressed most clearly in models where the average firing rate of the population is highly dependent on the amount of probability mass passing through an area of slow dynamics. (B) In the mesh algorithm, when cells become shear, probability mass which is pushed to the right due to incoming spikes also moves laterally (downwards) because it is spread evenly across each cell.

in the absence of incoming spikes. A mesh is constructed from strips which follow the trajectories of neurons in state space (Fig. 2.8). The trajectories form so-called characteristic curves of the neuron model from which this method is inspired (de Kamps, Lepperød, et al., 2019; de Kamps, 2013).

These trajectories are computed as part of a one-time preprocessing step using an appropriate integration technique and time step. Strips will often approach or recede from nullclines and stationary points and their width may shrink or expand according to their proximity to such elements. Each strip is split into cells. Each cell represents how far along the strip neurons will move in a single time step. As with the width of the strips, cells will become more dense or more sparse as the dynamics slow down and speed up respectively. The result of covering the state space with strips is a precomputed description of the model dynamics such that the state of a neuron in one cell of the mesh is guaranteed to be in the next cell along the strip after a single time step. Depending on the underlying neuron model, it can be difficult to get full coverage without cells becoming too small or shear. However, once built, the deterministic dynamics have effectively been “pre-solved” and baked into the mesh.

As with the grid algorithm, when the simulation is running, each cell is associated with a probability mass value which represents the probability of finding a neuron from the population with a state in that cell. When a probability density function (PDF) is defined across the mesh, computing the change to the PDF due to the deterministic dynamics of the neurons is simply a matter of shifting each cell’s probability mass value along its strip. In the C++ implementation, this requires no more than a pointer update and is, therefore, quicker than the grid algorithm for solving the deterministic dynamics as no transition matrix is applied to the cells.

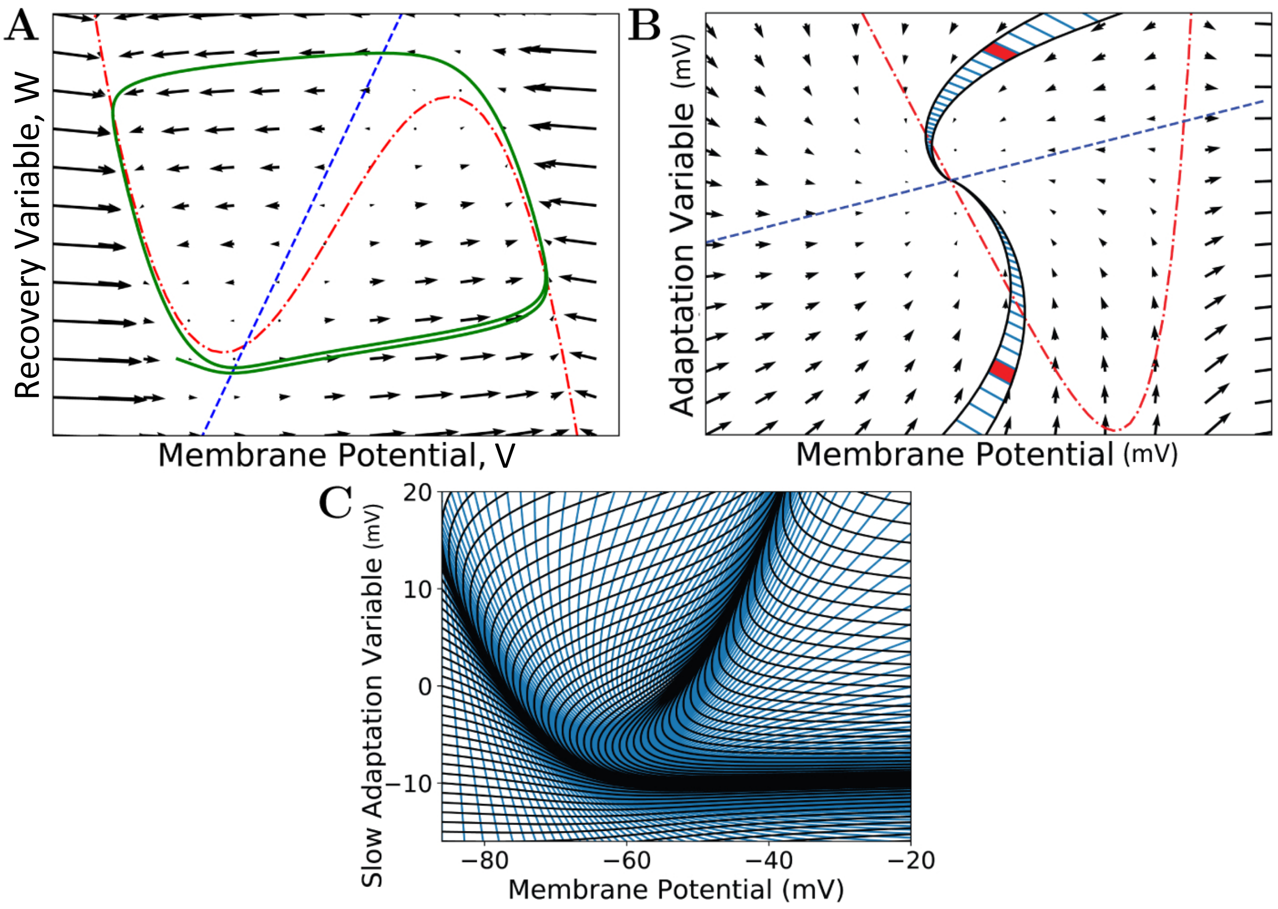


Figure 2.8: (A) A vector field of the FitzHugh-Nagumo neuron model (FitzHugh, 1961). Arrows show the direction of motion of states through the field according to the dynamics of the model. The red broken dashed nullcline indicates where the change in V is zero. The blue dashed nullcline indicates where the change in W is zero. The green solid line shows a potential path (trajectory) of a neuron in the state space. (B) A vector field for the adaptive exponential integrate-and-fire neuron model (Brette and Gerstner, 2005). Two strips are shown which follow the dynamics of the model and approach the stationary point where the nullclines cross. A strip is constructed between two trajectories in state space. Each time step of the two trajectories is used to segment the strip into cells. Because the strips approach a stationary point, they get thinner as the trajectories converge to the same point and cells get closer together as the distance in state-space travelled reduces per time step (neurons slow down as they approach a stationary point). Per time step, probability mass is shifted from one cell to the next along the strip. (C) The state-space of the Izhikevich simple neuron model (Izhikevich, 2003) which has been fully discretised into strips and cells.

Mesh algorithm does, however, still require a transition matrix to implement the effect of incoming spikes on the PDF. This transition matrix describes how the state of neurons in each cell is translated in the event of a single incoming spike. Unlike the grid algorithm, cells are unevenly distributed across the mesh and are different sizes and shapes. What proportion of probability mass is transferred to which cells with a single incoming spike is, therefore, different for all cells. During simulation, the total change in the PDF is calculated by shifting probability mass one cell down each strip and using the

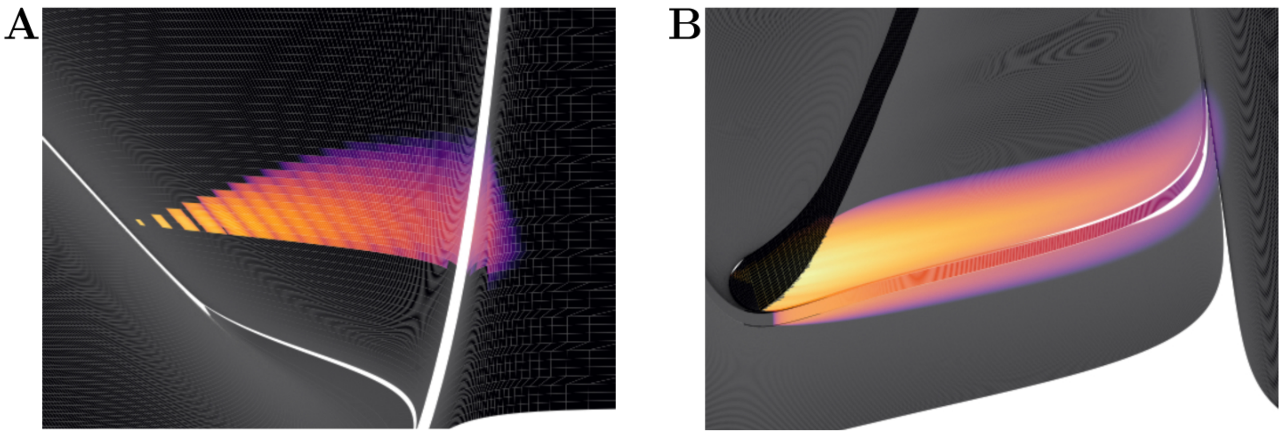


Figure 2.9: Heat plots for the probability density functions of two populations in MIIND. Brightness (more yellow) indicates a higher probability mass. Scales have been omitted as the underlying neuron models are arbitrary. (A) When the Poisson master equation is solved, probability mass is pushed to the right (higher membrane potential) in discrete steps. As time passes, the discrete steps are smoothed out due to the movement of mass according to the deterministic dynamics (following the strip). (B) A combination of mass travelling along strips and being spread across the state space by noisy input produces the behaviour of the population.

transition matrix to solve the master equation every time step. The combined effect can be seen in Fig. 2.9. The method of solving the master equation is explained in detail in de Kamps, 2013.

2.4.1 When not to use the Mesh Algorithm

Just as with the grid algorithm, certain neuron models are better suited to an alternative algorithm. In the mesh algorithm, very little error is introduced for the deterministic dynamics. Probability mass flows down each strip as it would without the discretisation and error is limited only to the size of the cells. When the master equation is solved, however, probability mass can spread to parts of state space which would see less or no mass. Fig. 2.7B demonstrates how in the mesh algorithm, as probability mass is pushed horizontally, very shear cells can allow mass to be incorrectly transferred vertically as well. In the grid algorithm, error is introduced in the opposite way. Solving the master equation pushes probability mass along horizontal rows of the grid and error is limited to the width of the row. The grid algorithm is preferable over the mesh algorithm for populations of neurons with one fast variable and one slow variable which can produce very shear cells in a mesh, e.g. in the Fitzhugh-Nagumo model (de Kamps, Lepperød, et al., 2019). In both algorithms, the error can be reduced by increasing the density of cells (by increasing the resolution of the grid, or by reducing the timestep and strip width of the mesh). However, better efficiency is achieved by using the appropriate algorithm.

2.4.2 Building a Mesh for the Mesh Algorithm

Before a simulation can be run for a population which uses the mesh algorithm, the pre-calculation steps of generating a mesh and transition matrices must be performed. Fig. 2.10 shows the full pre-processing pipeline for mesh algorithm. The mesh is a collection of strips made up of quadrilateral cells. As mentioned earlier, probability mass moves along a strip from one cell to the next each time step which describes the deterministic dynamics of the model. Defining the cells and strips of a 2D mesh is not generally a fully automated process and the points of each quadrilateral must be defined by the mesh developer and stored in a *.mesh* file. When creating the mesh, the aim is to cover as much of the state space as possible without allowing cells to get too small or misshapen. An example of a full mesh generation script for the Izhikevich simple neuron model (Izhikevich, 2003) is available in section B.9.1 of the supplementary material. MIIND provides *miind.miind_api.LifMeshGenerator*, *miind.miind_api.QifMeshGenerator*, and *miind.miind_api.EifMeshGenerator* scripts to automatically build the 1D leaky integrate-and-fire, quadratic integrate-and-fire, and exponential integrate-and-fire neuron meshes respectively. They can be called from the CLI. The scripts generate the three output files which any mesh generator script must produce: a *.mesh* file, a *.stat* file which defines extra cells in the mesh to hold probability mass that has settled at a stationary point, and a *.rev* file which defines a "reversal mapping" indicating how probability mass is transferred from strips in the mesh to the stationary cells. More information on *.mesh*, *.stat*, and *.rev* files is provided in the supplementary material section B.6.

Once the *.mesh*, *.stat*, and *.rev* files have been generated by the user or by one of the automated 1D scripts, the Python command line interface, *miind.miindio*, provides commands to convert the three files into a single *.model* file and generate transition matrices stored in *.mat* files. The model file is what will be referenced and read by MIIND to load a mesh for a simulation. To generate this file, use the CLI command, **generate-model**. The command parameters are shown in Table 2.2. All input files must have the same base name, for example, *lif.mesh*, *lif.stat*, and *lif.rev.* If the command runs successfully, a new file will be created: *basename.model.* A number of pre-generated models are available in the *examples* directory of the MIIND repository to be used "out of the box" including the adaptive exponential integrate-and-fire and conductance-based neuron models.

Listing 2.7: Generate a Model in the CLI

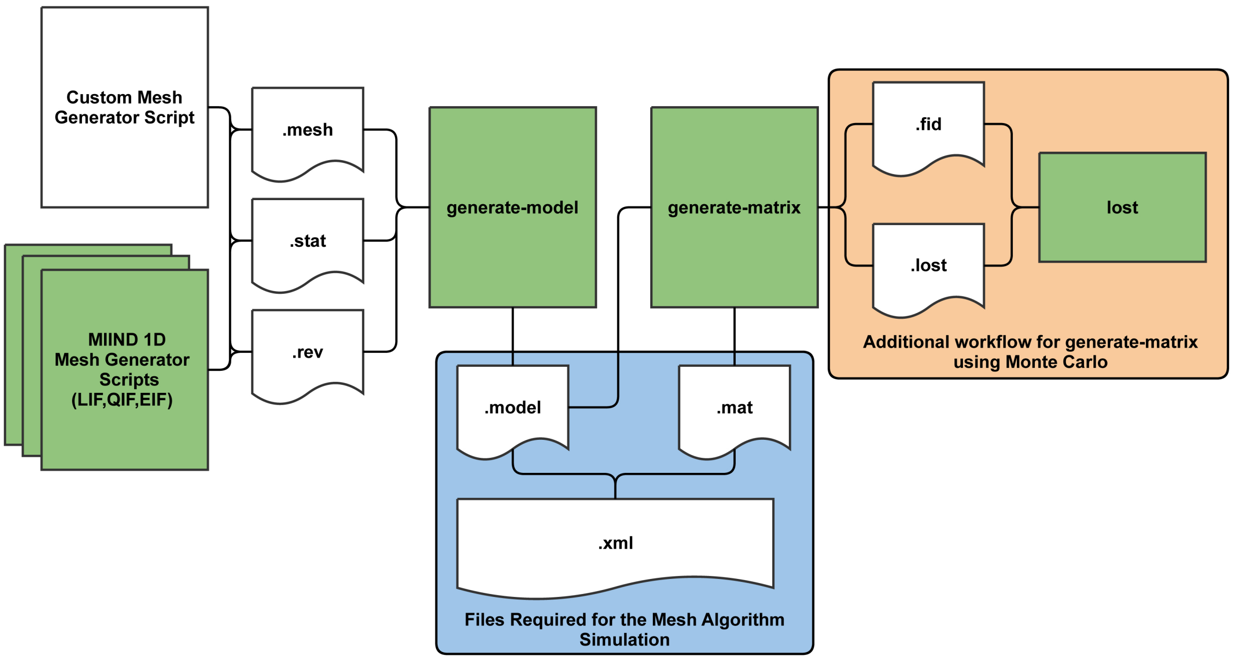

Custom Mesh Generator Script
MIIND 1D Mesh Generator Scripts (LIF,QIF,EIF)
.mesh
.stat
.rev
generate-model
generate-matrix
.fid
.lost
lost
Additional workflow for generate-matrix using Monte Carlo
.model
.mat
.xml
Files Required for the Mesh Algorithm Simulation

Figure 2.10: The MIIND processes and generated files required at each stage of pre-processing for the mesh algorithm. The shaded green rectangles represent automated processes run via the MIIND CLI.

Table 2.2: Parameters for the **generate-model** command in the CLI.

Parameter Name	Notes
basename	The shared name of the *.mesh*, *.stat*, *.rev* and generated *.model* files.
reset	The value (usually representing membrane potential) which probability mass will be transferred to having passed the threshold.
threshold	The value (usually representing membrane potential) beyond which probability mass will be transferred to the reset value.

```
> generate-model lif -60.0 -30.0
```

The generated *.model* file contains the mesh vertices, some summary information such as the time step used to generate the mesh and the threshold and reset values, and mapping of threshold cells to reset cells.

In the mesh algorithm, transition matrices are used to solve the Poisson master equation which describes the movement of probability mass due to incoming random spikes. In the mesh algorithm, one transition matrix is required for each post synaptic efficacy that will be needed in the simulation. So if a population is going to receive spikes which cause jumps of 0.1mV and 0.5mV, two transition matrices are required. It is demonstrated later how the efficacy can be made dependent on the membrane potential or other variables. Each transition matrix is stored in a *.mat* file and contains a list of source cells, target cells, and proportions of probability mass to be transferred to each. For a given cell in the mesh, neurons with a state inside that cell which receive a single external spike will shift their location in state space by the value of the efficacy. Neurons from the same cell could therefore end up in many other different cells, though often ones which are nearby. It is assumed that neurons are distributed uniformly across the source cell. Therefore, the proportion of neurons which end up in each of the other cells can be calculated. MIIND performs this calculation in two ways, the choice for which is given to the user.

The first method is to use a Monte Carlo approach such that a number of points are randomly placed in the source cell then translated according to the efficacy. A search takes place to find which cells the points were translated to and the proportions are calculated from the number of points in each. For many meshes, a surprisingly small number of points, around 10, is required in each cell to get a good approximation for the transition matrix and the process is therefore quite efficient. As shown in Fig. 2.10, an additional process is required when generating transition matrices using Monte Carlo which includes two further intermediate files, *.fid* and *.lost*. All points must be accounted for when performing the search and in cases where points are translated outside of the mesh, an exhaustive search must be made to find the closest cell. The **lost** command allows the user to speed up this process which is covered in detail in section B.6.1 of the supplementary material.

The second method translates the actual vertices of each cell according to the efficacy and calculates the exact overlapping area with other cells. The method by which this is achieved is the same as that used to generate the transition matrix of the grid algorithm, described in section 2.3.1. This method provides much higher accuracy than Monte Carlo but is one order of magnitude slower (it takes a similar amount of time to perform Monte Carlo with 100 points per cell). For some meshes, it is crucial to include very small transitions between cells to properly capture the dynamics which justifies the need for the slower method. It also benefits from requiring no additional user input in contrast to the Monte Carlo method.

In *miind.miindio*, the command **generate-matrix** can be used to automatically generate each *.mat* file. In order to work, there must be a *basename.model* file in the working directory. The **generate-matrix** command takes six parameters which are described in Table 2.3. Listing 2.8 shows an example of the **generate-matrix** command. If successful, two files are generated: *basename.mat* and *basename.lost*.

Table 2.3: Parameters for the **generate-matrix** command in the CLI.

Parameter Name	Notes
basename	The shared name of the *.model*, *.fid* (if required), and generated *.mat* files.
v_efficacy	The efficacy value in the v (membrane potential) direction. If the parameter *h_efficacy* is used, this should be zero.
points / precision	For Monte Carlo, this gives the number of points per cell to use for approximating the transition matrix. For the geometric method, transitions are stored in the *.mat* file to the nearest $\frac{1}{precision}$
h_efficacy	The efficacy value in the h direction. If the parameter *v_efficacy* is used, this should be zero.
reset-shift	The shift in the h direction which neurons take when being reset.
use_geometric	A boolean flag set to "true" if the geometric method is used and "false" for Monte Carlo.

Listing 2.8: The *miind.miindio* command to generate a matrix using the adex.model file with an efficacy of 0.1 in v and a jump of 5.0 in w when a neuron spikes. The Monte Carlo method has been chosen with 10 points per cell.

```
> generate-matrix adex 0.1 10 0.0 5.0 false
```

Once **generate-matrix** has completed, a *.mat* file will have been generated and the *.model* file will have been amended to include a <*Reset Mapping*> section. Similar to the reversal mapping in the *.rev* file, the reset mapping describes movement of probability mass from the cells which lie across the threshold potential to cells which lie across the reset potential. If the threshold or reset values are changed but no other change is made to the mesh, it can be helpful to re-run the mapping calculation without having to completely re-calculate the transition matrix. *miind.miindio* provides the command **regenerate-reset** which takes the base name and any new reset shift value (0 if not required) as parameters. This will quickly replace the reset mapping in the *.model* file.

Listing 2.9: The user may change the <*Threshold*> and <*Reset*> values in the *.model* file (or re-call **generate-model** with different threshold and reset values) then update the existing Reset Mapping. In this case, the adex.model was updated with a reset w shift value of 7.0.

```
> regenerate-reset adex 7.0
```

With all required files generated, a simulation using the mesh algorithm can now be run in MIIND.

2.4.3 Jump Files

In some models, it is helpful to be able to set the efficacy as a function of the state. For example, to approximate adaptive behaviour where the post-synaptic efficacy lowers as the membrane potential increases. Jump files have been used in MIIND to simulate the Tsodyks-Markram (Tsodyks and Markram, 1997) synapse model as described by de Kamps, Lepperød, et al., 2019. In the model, one variable/dimension is required to represent the membrane potential, V, of the post-synaptic neuron and the second to represent the synaptic contribution, G. G and V are then used to derive the post-synaptic potential caused by an incoming spike. Before generating the transition matrix, each cell can be assigned its own efficacy for which the transitions will be calculated. During generation, Monte Carlo points will be translated according to that value instead of a constant across the entire mesh. When calling the **generate-matrix** command, a separate set of three parameters is required to use this feature. The base name of the model file, the number of Monte Carlo points per cell, and a reference to a *.jump* file which stores the efficacy values for each cell in the mesh.

Listing 2.10: Generate a transition matrix with a jump file in the CLI

```
> generate-matrix adex 10 adex.jump
```

As with the files required to build the mesh, the jump file must be user-generated as the efficacy values may be non-linear and involve one or both of the dimensions of the model. The format of a jump file is shown in listing 2.11. The <*Efficacy*> element of the XML file gives an efficacy value for both dimensions of the model and is how the resulting transition matrix will be referenced in the simulation. The <*Translations*> element lists the efficacy in both dimensions for each cell in the mesh.

Listing 2.11: The format of the jump file. Each line in the <*Translations*> block gives the strip,cell coordinates of the cell followed by the h efficacy then the v efficacy. The <*Efficacy*> element gives a reference efficacy which will be used to reference the transition matrix built with this jump file. It must therefore be unique among jump files used for the same model.

```
<Jump>
<Efficacy>0.0 0.1</Efficacy>
<Translations>
0,0   0.0   0.1
1,0   0.0   0.1
1,1   0.0   0.10012
1,2   0.0   0.10045
...
```

```
</Translations>
</Jump>
```

After calling **generate-matrix**, as before, the *.mat* file will be created with the quoted values in the <*Efficacy*> element of the jump file. As with the vanilla Monte Carlo generation, the additional process of tracking lost points must be performed.

2.5 Writing the XML File

MIIND provides an intuitive XML-style language to describe a simulation and its parameters. This includes descriptions of populations, neuron models, integration techniques, and connectivity as well as general parameters such as time step and duration. The XML file is split into sections which are sub-elements of the XML root node, <*Simulation*>. They are Algorithms, Nodes, Connections, Reporting, and SimulationRunParameter. These elements make up the major components of a MIIND simulation.

2.5.1 Algorithms

An <*Algorithm*> in the XML code describes the simulation method for a population in the network. The nodes of the network represent separate instances of these algorithm elements. Therefore, many nodes can use the same algorithm. Each algorithm has different parameters or supporting files but as a minimum, all algorithms must declare a type and a name. Each algorithm is also implicitly associated with a "weight type". All algorithms used in a single simulation must be compatible with the weight type as it describes the way that populations interact. The <*WeightType*> element of the XML file can take the values, "double", "DelayedConnection", or "CustomConnectionParameters". Which value the weight type element takes influences which algorithms are available in the simulation and how the connections between populations will be defined. The following sections cover all Algorithm types currently supported in MIIND. Table 2.4 lists these algorithms and their compatible weight types.

RateAlgorithm

RateAlgorithm is used to supply a Poisson distributed input (with a given average firing rate) to other nodes in the simulation. It is typically used for simulating external input. The <*rate*> sub-element is used to define the activity value which is usually a firing rate.

Table 2.4: Compatible weight types for each algorithm type defined in the simulation XML file.

Algorithm Name	double	DelayedConnection	CustomConnectionParameters
RateAlgorithm	✓	✓	✓
MeshAlgorithm		✓	
MeshAlgorithmCustom			✓
GridAlgorithm			✓
GridJumpAlgorithm			✓
OUAlgorithm		✓	
WilsonCowanAlgorithm	✓		
RateFunctor	✓	✓	✓

Listing 2.12: A RateAlgorithm definition with a constant rate of 100Hz.

```
<Algorithm name="Cortical Background Algorithm" type="RateAlgorithm">
   <rate>100.0</rate>
</Algorithm>
```

MeshAlgorithm and MeshAlgorithmCustom

In section 2.4.2, we saw how to generate *.model* and *.mat* files. These are required to simulate a population using the mesh algorithm. Algorithm type=MeshAlgorithm tells MIIND to use this technique. The model file is referenced as an attribute to the Algorithm definition. The *TimeStep* child element must match that which was used to generate the mesh. This value is quoted in the model file. As many *MatrixFile* elements can be declared as are required for the simulation, each with an associated .mat file reference.

Listing 2.13: A MeshAlgorithm definition with two matrix files.

```
<Algorithm type="MeshAlgorithm" name="ALG_ADEX" modelfile="adex.model" >
   <TimeStep>0.001</TimeStep>
   <MatrixFile>adex_0.05_0_0_0_.mat</MatrixFile>
   <MatrixFile>adex_-0.05_0_0_0_.mat</MatrixFile>
</Algorithm>
```

MeshAlgorithm provides two further optional attributes in addition to *modelfile*. The first is *tau_refractive* which enables a refractory period and the second is *ratemethod* which takes the value "AvgV" if the activity of the population is to be represented by the average membrane potential. Any other value for *ratemethod* will set the activity to the default average firing rate. The activity value is what will be passed to other populations in the network as well as what will be recorded as the activity for any populations using this algorithm.

When the weight type is set to CustomConnectionParameters, the type of this algorithm definition should be changed to MeshAlgorithmCustom. No other changes to the definition are required.

GridAlgorithm and GridJumpAlgorithm

For populations which use the grid algorithm, the following listing is required. Similar to the MeshAlgorithm, the model file is referenced as an attribute. However, there are no matrix files required as the transition matrix for solving the Poisson master equation is calculated at run time. The transition matrix for the deterministic dynamics, stored in the *.tmat* file, is referenced as an attribute as well. Attributes for *tau_refractive* and *ratemethod* are also available with the same effects as for MeshAlgorithm.

Listing 2.14: A GridAlgorithm definition using the AvgV (membrane potential) rate method.

```
<Algorithm type="GridAlgorithm" name="GRIDALG_FN" modelfile="fn.model" tau_refractive
    ↪ ="0.0" transformfile="fn_0_0_0_0_.tmat" start_v="-1.0" start_w="-0.3"
    ↪ ratemethod="AvgV">
<TimeStep>0.00001</TimeStep>
</Algorithm>
```

GridAlgorithm also provides additional attributes *start_v* and *start_w* which allows the user to set the starting state of all neurons in the population which creates an initial probability mass of 1.0 in the corresponding grid cell at the start of the simulation.

GridJumpAlgorithm provides similar functionality as MeshAlgorithm when the transition matrix is generated using a jump file. That is, the efficacy applied to each cell when calculating transitions differs from cell to cell. In GridJumpAlgorithm, the efficacy at each cell is multiplied by the distance between the central v value of the cell and a user-defined “stationary” value. The initial efficacy and the stationary values are defined by the user in the XML <*Connection*> elements. GridJumpAlgorithm is useful for approximating populations of neurons with a voltage-dependent synapse.

Listing 2.15: A GridJumpAlgorithm definition and corresponding Connection with a “stationary” attribute. The efficacy at each grid cell will equal the original efficacy value (-0.05) multiplied by the difference between each cell’s central v value and the given stationary value (-65)

```
<Algorithm type="GridJumpAlgorithm" name="ALG_ADEX" modelfile="adex.model"
    ↪ tau_refractive="0.0" transformfile="adex_0_0_0_0_.tmat" start_v="-65.0"
    ↪ start_w="0.0">
<TimeStep>0.0001</TimeStep>
</Algorithm>
...
<Connection In="BG_NOISE" Out="ADEX_NODE" num_connections="1" efficacy="-0.05" delay=
    ↪ "0.0" stationary="-65.0"/>
```

Additional Algorithms

MIIND also provides OUAlgorithm and WilsonCowanAlgorithm. The OUAlgorithm generates an Ornstein–Uhlenbeck process (Uhlenbeck and Ornstein, 1930) for simulating a population of LIF neurons. The WilsonCowanAlgorithm implements the Wilson-Cowan model for simulating population activity (H. Wilson and Cowan, 1972). Examples of these algorithms are provided in the examples directory of the MIIND repository (*examples/twopop* and *examples/model_archive/WilsonCowan*).

One final algorithm, RateFunctor, behaves similarly to RateAlgorithm. However, instead of a rate value, the child value defines the activity using a C++ expression in terms of variable, t, representing the simulation time.

Listing 2.16: A RateFunctor algorithm definition in which the firing rate linearly increases to 100Hz over 0.1 seconds and remains at 100Hz thereafter.

```
<Algorithm type="RateFunctor" name="ExternalInput">
   <expression><![CDATA[ t < 0.1 ? (t/0.1)*100 : 100 ]]></expression>
</Algorithm>
```

A CDATA expression is not permitted when using MIIND in Python or when calling *miind.run*. However, RateFunctor can still be used with a constant expression (although this has no benefit beyond what RateAlgorithm already provides). CDATA should only be used when MIIND is built from source (not installed using pip) and the MIIND API is used to generate C++ code from an XML file.

2.5.2 Nodes

The <*Node*> block lists instances of the Algorithms defined above. Each node represents a single population in the network. To create a node, the user must provide the name of one of the algorithms defined in the algorithm block which will be instantiated. A name must also be given to uniquely identify this node. The type describes the population as wholly inhibitory, excitatory, or neutral. The type dictates the sign of the post synaptic efficacy caused by spikes from this population. Setting the type to neutral allows the population to produce both excitatory and inhibitory (positive and negative) synaptic efficacies. For most algorithms, the valid types for a node are *EXCITATORY*, *INHIBITORY*, and *NEUTRAL*. *EXCITATORY_DIRECT* and *INHIBITORY_DIRECT* are also available but mean the same as *EXCITATORY* and *INHIBITORY* respectively.

Listing 2.17: Three nodes defined in the Nodes section using the types *NEUTRAL*, *INHIBITORY*, and *EXCITATORY* respectively.

```
<Nodes>
...
<Node algorithm="GRIDALG_FN" name="POP_1" type="NEUTRAL" />
<Node algorithm="ALG_ADEX" name="ADEX_NODE" type="INHIBITORY" />
<Node algorithm="RATEFUNC_BACKGROUND" name="BG_NOISE" type="EXCITATORY" />
...
</Nodes>
```

Many nodes can reference the same algorithm to use the same population model but they will behave independently based on their individual inputs.

2.5.3 Connections

The connections between the nodes are defined in the <*Connections*> sub-element. Each connection can be thought of as a conduit which passes the output activity from the "In" population node to the "Out" population node. The format used to define the connections is dependent on the choice of *WeightType*. When the type is *double*, connections require a single value which represents the connection weight. This will be multiplied by the output activity of the In population and passed to the Out population. The sign of the weight must match the In node's type definition (*EXCITATORY*, *INHIBITORY*, *NEUTRAL*).

Listing 2.18: A simple double WeightType Connection with a single rate multiplier.

```
<WeightType>double</WeightType>
<Connections>
...
<Connection In="RATEFUNC_BACKGROUND" Out="WC_POP">0.1</Connection>
...
</Connections>
```

Many algorithms use the *DelayedConnection* weight type which requires three values to define each connection. The first is the number of incoming connections each neuron in the Out population receives from the In population. This number is effectively a weight and is multiplied by the output activity of the In population. For example, if the output firing rate of an In population is 10Hz and the number of incoming connections is set to 10, the effective average incoming spike rate to each neuron in the Out population will be 100 Hz. The second value is the post synaptic efficacy whose sign must match the type of the In population. If the Out population is an instance of MeshAlgorithm, the efficacy must also match one of the provided *.mat* files. The third value is the connection delay in seconds. The delay is implemented in the same way as the refractory period in the mesh and grid algorithms. The output activity of the In population is placed at the beginning of the queue and shifted towards the end of the queue over subsequent iterations. The input to the Out population is taken as the linear interpolation between the final two values in the queue.

Listing 2.19: A DelayedConnection with number of connections = 10, efficacy = 0.1, and delay of 1ms.

```
<WeightType>DelayedConnection</WeightType>
<Connections>
...
<Connection In="RATEFUNC_BACKGROUND" Out="BURSTER">10 0.1 0.001</Connection>
...
</Connections>
```

With the addition of GridAlgorithm, there was a need for a more flexible connection type which would allow custom parameters to be applied to each connection. When using the *CustomConnectionParameters* weight type, the key-value attributes of the connections are passed as strings to the C++ implementation. By default, custom connections require the same three values as *DelayedConnection*: *num_connections*, *efficacy*, and *delay*. *CustomConnectionParameters* can therefore be used with mesh algorithm nodes as well as grid algorithm nodes although MeshAlgorithm definitions must have the type attribute set to MeshAlgorithmCustom instead.

Listing 2.20: A MeshAlgorithmCustom definition for use with WeightType=CustomConnectionParameters and a Connection using the num_connections, efficacy, and delay attributes.

```
<WeightType>CustomConnectionParameters</WeightType>

<Algorithms>
...
<Algorithm type="MeshAlgorithmCustom" name="ALG_ADEX" modelfile="adex.model" >
   <TimeStep>0.001</TimeStep>
   <MatrixFile>adex_0.05_0_0_0_.mat</MatrixFile>
   <MatrixFile>adex_-0.05_0_0_0_.mat</MatrixFile>
</Algorithm>
...
</Algorithms>


<Connections>
...
<Connection In="ALG_ADEX" Out="RG_E" num_connections="1" efficacy="0.05" delay="0.0"/
    ↪ >
...
</Connections>
```

Other combinations of attributes for connections using CustomConnectionParameters are available for use with specific specialisations of the grid algorithm which are discussed in section B.4 of the supplementary material. Any number of attributes are permitted but they will only be used if there is an algorithm specialisation implemented in the MIIND code base.

2.5.4 SimulationRunParameter

The <*SimulationRunParameter*> block contains parameter settings for the simulation as a whole. The sub-elements listed in Table 2.5 are required for a full definition. Although most of the sub-elements are self-explanatory, *t_step* has the limitation that it must match or be an integer multiple of all time steps defined by any MeshAlgorithm and GridAlgorithm instances. *master_steps* is used only for the GPGPU implementation of MIIND (section 2.6). It allows the user to set the number of Euler iterations per time step to solve the master equation. By default, the value is 10. However, to improve accuracy or to avoid blow-up in the case where the time step is too large or the local dynamics are unstable, *master_steps* should be increased.

Table 2.5: The required sub-elements for the *SimulationRunParameter* section of the XML simulation file.

Element	Notes
SimulationName	The name of the simulation.
t_end	The simulation end time.
t_step	The time step of the simulation.
name_log	A file name for logging. The file is stored in the output directory of the simulation.
master_steps	The number of Euler iterations per time step used to solve the master equation in the GPGPU implementation.

2.5.5 Reporting

The <*Reporting*> block is used to describe how the output is displayed and recorded from the simulation. There are three ways to record output from the simulation: Density, Rate, and Display. The <*Rate*> element takes the node *name* and *t_interval* as attributes and creates a single file in the output directory. *t_interval* must be greater than or equal to the simulation time step. At each *t_interval* of the simulation, the output activity of the population is recorded on a new line of the generated file. Although the element is called "Rate", if average membrane potential has been chosen as the activity of this population, this is what will be recorded here. <*Density*> is used to record the full probability density of the given population node. As density is only relevant for the population density technique, it can only be recorded from nodes which instantiate the mesh or grid algorithm types. The attributes are the node *name*, *t_start*, *t_end*, and *t_interval* which define the simulation times to start and end recording the density at the given interval. A file which holds the probability mass values for each cell in the mesh or grid will be created in the output directory for each *t_interval* between *t_start* and *t_end*. Finally, the <*Display*> element can be used to observe the evolution of the probability density function as the simulation is running. If a *Display* element is added in the XML file for a specific node, when the simulation is run, a graphical window will open and display the probability density for each time step. Again, display is only applicable to algorithms involving densities. Enabling the display can significantly slow the simulation down. However, it is useful for debugging the simulation and furthermore, each displayed frame is stored in the output directory so that a movie can be made of the node's behaviour. How to generate this movie is discussed later in section 2.8.1.

Listing 2.21: A set of reporting definitions to record the probability densities and rates of two populations, S and D. The densities will also be displayed during simulation.

```
<Reporting>
...
   <Density node="S" t_start="0.0" t_end="6.0" t_interval="0.01" />
   <Density node="D" t_start="0.5" t_end="1.5" t_interval="0.001" />
   <Display node="S" />
   <Display node="D" />
   <Rate node="S" t_interval="0.0001" />
   <Rate node="D" t_interval="0.0001" />
...
</Reporting>
```

2.5.6 Variables

The <*Simulation*> element can contain multiple <*Variable*> sub-elements each with a unique name and value. Variables are provided for the convenience of the user and can replace any values in the XML file. For example, a variable named *TIME_END* can be defined to replace the value in the *t_end* element of the *SimulationRunParameter* block. When the simulation is run, the value of *t_end* will be replaced with the default value provided in the Variable definition. Using variables makes it easy to perform parameter sweeps where the same simulation is run multiple times and only the variable's value is changed. How parameter sweeps are performed is covered in the supplementary material section B.8. All values in a MIIND XML script can be set with a variable name. The type of the Variable is implicit and an error will be thrown if, say, a non-numerical value is passed to the tau_refractive attribute of a MeshAlgorithm object.

Listing 2.22: A Variable definition. TIME_END has a default value of 18.0 and is used in the *t_end* parameter definition.

```
<Variable Name='TIME_END'>18.0</Variable>
...
<t_end>TIME_END</t_end>
```

2.6 MIIND on the GPU

The population density techniques of the mesh and grid algorithms rely on multiple applications of the transition matrix which can be performed on each cell in parallel. This makes the algorithms prime candidates for parallelisation on the graphics card. In the CPU versions, the probability mass is stored in separate arrays, one for each population/node in the simulation. For the GPGPU version, these are concatenated into one large probability mass vector so all cells in all populations can be processed in parallel. From the user's perspective, switching between CPU and GPU implementations is trivial. In the XML file for a simulation which uses MeshAlgorithm or GridAlgoirithm, to switch to the vectorised GPU version, the Algorithm types must be changed to MeshAlgorithmGroup and GridAlgorithmGroup. All other attributes remain the same. Only MeshAlgorithmGroup, GridAlgorithmGroup, and RateFunctor/RateAlgorithm types can be used for a vectorised simulation. When running a MIIND simulation containing a group algorithm from a Python script, instead of importing *miind.miindsim*, *miind.miindsimv* should be used. The Python module *miind.run* is agnostic to the use of group algorithms so can be used as shown previously.

Listing 2.23: A MeshAlgorithmGroup definition is identical to a MeshAlgorithm definition except for the type.

```
<Algorithm type="MeshAlgorithmGroup" name="ALG_ADEX" modelfile="adex.model" >
   <TimeStep>0.001</TimeStep>
   <MatrixFile>adex_0.05_0_0_0_.mat</MatrixFile>
   <MatrixFile>adex_-0.05_0_0_0_.mat</MatrixFile>
</Algorithm>
<Algorithm type="GridAlgorithmGroup" name="OSC" modelfile="fn.model" tau_refractive="
   ↪ 0.0" transformfile="fn_0_0_0_0_.tmat" start_v="-1.0" start_w="-0.3" ratemethod
   ↪ ="AvgV">
<TimeStep>0.00001</TimeStep>
</Algorithm>
```

The GPGPU implementation uses the Euler method to solve the master process during each iteration. It is, therefore, susceptible to blow-up if the time step is large or if the local dynamics of the model are stiff. The user has the option to set the number of Euler steps taken each iteration using the *master_steps* value of the SimulationRunParameter block in the XML file. A higher value reduces the likelihood of blow-up but increases the simulation time.

In order to run the vectorised simulations, MIIND must be running on a CUDA-enabled machine and have CUDA enabled in the installation (CUDA is supported in the Windows and Linux python installations). Section B.3 in the supplementary material goes into greater detail about the systems architecture differences between the CPU and GPU versions of the MIIND code. Using the "Group" algorithms is recommended if possible as it provides a significant performance increase. As shown in de Kamps, Lepperød, et al., 2019, with the use of the GPGPU, a population of conductance-based neurons in MIIND performs comparably to a NEST simulation of 10000 individual neurons but using an order of magnitude less memory. This allows MIIND to simulate many thousands of populations on a single PC.

2.7 Running a MIIND Simulation in Python

As demonstrated in the quick start guide, the command **python -m miind.run** takes a simulation XML file as a parameter and runs the simulation. A similar script may be written by the user to give more control over what happens during a simulation and how output activity is recorded and processed. It even allows MIIND simulations to be integrated into other Python applications such as TVB

(Sanz Leon et al., 2013) so the population density technique can be used to solve the behaviour of nodes in a brain-scale network (see section 2.10). To run a MIIND simulation in a Python script, the module *miind.miindsim* must be imported (or *miind.miindsimv* if the simulation uses MeshAlgorithmGroup or GridAlgorithmGroup and therefore requires CUDA support). Listing 2.24 shows an example script which uses the following available functions to control the simulation.

Listing 2.24: A simple python script for running a MIIND simulation and plotting the results.

```python
import matplotlib.pyplot as plt
import miind.miindsim as miind

miind.init(1, "lif.xml")

timestep = miind.getTimeStep()
simulation_length = miind.getSimulationLength()
print('Timestep from XML : {}'.format(timestep))
print('Sim time from XML : {}'.format(simulation_length))

miind.startSimulation()

constant_input = [2500]
activities = []
for i in range(int(simulation_length/timestep)):
    activities.append(miind.evolveSingleStep(constant_input)[0])

miind.endSimulation()

plt.figure()
plt.plot(activities)
plt.title("Firing Rate.")

plt.show()
```

2.7.1 init(node_count,simulation_xml_file,...)

The *init* function should be called first once the MIIND library has been imported. This sets up the simulation ready to be started. The *node_count* parameter allows for multiple instantiations of the simulation to be run simultaneously. The Nodes, Connections, and Reporting blocks from the simulation file will be duplicated, effectively running the same model *node_count* times simultaneously in the same simulation. This functionality was included to allow TVB to run the simulation defined in the XML file multiple times (see section 2.10). The *simulation_xml_file* parameter gives the name of the simulation XML file to be run. If the file has any variables defined, these are made available in Python as additional parameters to the *init* function. In this way, the use of XML variables can be used for parameter sweeps. All variables must be passed as strings. If a variable is not set in the call to *init*, the default value defined in the XML file will be used.

Listing 2.25: Calling init for a MIIND simulation lif.xml with the Variable SIM_TIME set to 0.4.

```
miind.init(1, "lif.xml", SIM_TIME="0.4")
```

2.7.2 getTimeStep() and getSimulationLength()

Once *init* has been called, the functions *getTimeStep* and *getSimulationLength* can be used to extract the time step and simulation length in seconds from the simulation respectively. The Python script controls when each iteration of the MIIND simulation is called and so it needs to know the total number of iterations to make. Furthermore, it can be useful for integration with other systems to know these values.

2.7.3 startSimulation()

startSimulation indicates in the Python script that the simulation should be initialised ready for the simulation loop to be called.

2.7.4 evolveSingleStep(input)

By calling *evolveSingleStep* in the Python script, the MIIND simulation will move forward one time step. This function takes a list of numbers as a parameter. The list corresponds to inputs to the population nodes in the MIIND simulation. In this way, the user may control the behaviour of the simulation from the Python script during the simulation. The *evolveSingleStep* function also returns a list of numbers which are the output activities of the population nodes. Section 2.7.6 provides more information about how to use the input and output of this function. *evolveSingleStep* should be called in a loop which will run the same number of iterations as would be expected if the XML file were run in MIIND directly, that is, the simulation length divided by the time step.

2.7.5 endSimulation()

It is good practice to call *endSimulation* once all iterations of the simulation have been performed. This allows MIIND to clean up and to print the performance statistics to the console.

2.7.6 Additional XML Code for Python Support

Although it is still possible to use *RateFunctor* or *RateAlgorithm* to set input rates to populations in a Python MIIND simulation, *evolveSingleStep()* provides a means to pass the input rates as a parameter so that more complex input patterns can be used. In order to indicate that a population will receive input externally from the Python script (via the list input to *evolveSingleStep()*) a special connection type must be defined in the <*Connections*> section of the XML.

Listing 2.26: Special connection types for use in Python.

```
<Connections>
...
<IncomingConnection Node="E">1 0.01 0</IncomingConnection>
<OutgoingConnection Node="E"/>
...
</Connections>
```

Listing 2.26 defines an input to node E which will be interpreted as a DelayedConnection with the number of connections equal to 1 and a post-synaptic efficacy of 0.01. No delay is defined here although it is permitted. OutgoingConnections are used to declare which nodes in the population network will pass their activity back to the Python script after each iteration. If the two connections in the listing are the only instances of IncomingConnection and OutgoingConnection, then the *evolveSingleStep* function will expect as a parameter, a list with one numeric value to represent the incoming rate to node E. *evolveSingleStep* will return a list with a single numeric value representing the activity of node E. In cases where there is more than one IncomingConnection, the order of values in the Python list parameter to *evolveSingleStep* is the same as the order of IncomingConnections defined in the XML. Similarly, with OutgoingConnections, the order of the list of activities returned from *evolveSingleStep* is the same as the order of declaration in the XML file.

2.8 Using the CLI to Quickly View Results

Once a simulation has been run, either using *miind.run* or from a user-written Python script, the *miind.miindio* CLI can be used to quickly plot the recorded results. As mentioned, the commands used in *miindio* are based on the module *miind.miind_api* and are reproducible in a Python script. However, it can be convenient to be able to run them directly from the command line to aid fast prototyping and bug fixing of models and simulations. The following section lists some common commands in the

CLI and their usage. The accompanying files for this example are in the *examples/cli_plots* directory. The following command starts the CLI and presets the user with a prompt:

Listing 2.27: Run the CLI.

```
$ python -m miind.miindio
```

When *miind.miindio* is called for the first time in a working directory, the user must identify the XML file which will describe the current working simulation. MIIND stores a reference to this file in a settings file in the working directory so that all subsequent commands will reference this simulation. Even if *miind.miindio* is quit and restarted, the current working simulation will be used as the context for commands until a new current working simulation is defined or if it is called in a different directory. The user can set the current working simulation with the **sim** command.

Listing 2.28: Load a simulation file in the CLI.

```
> sim example.xml
```

Calling **sim** without a parameter will list information about the current working simulation such as the output directory, XML file name and provide a list of the defined variables and nodes.

During the simulation, MIIND generates output files according to the requirements of the <*Recording*> object of the XML file which could include the average firing rate of population nodes or their densities at each time interval. The average firing rate can be plotted from the CLI using the **rate** command followed by the name of the population node. To be reminded of the node names, the user can call **sim** or **rate** without parameters.

Listing 2.29: Plot the rate of population POP1 in the CLI.

```
> rate POP1
```

Even while a simulation is running, calling **rate** in the CLI will plot the recorded activity up to the latest simulated time point. This is useful to keep an eye on the simulation as it progresses without waiting for completion. An example of the plots produced by **rate** is shown in Fig. 2.11A.

For populations using the grid or mesh algorithms, the user can call the **plot-density** command with

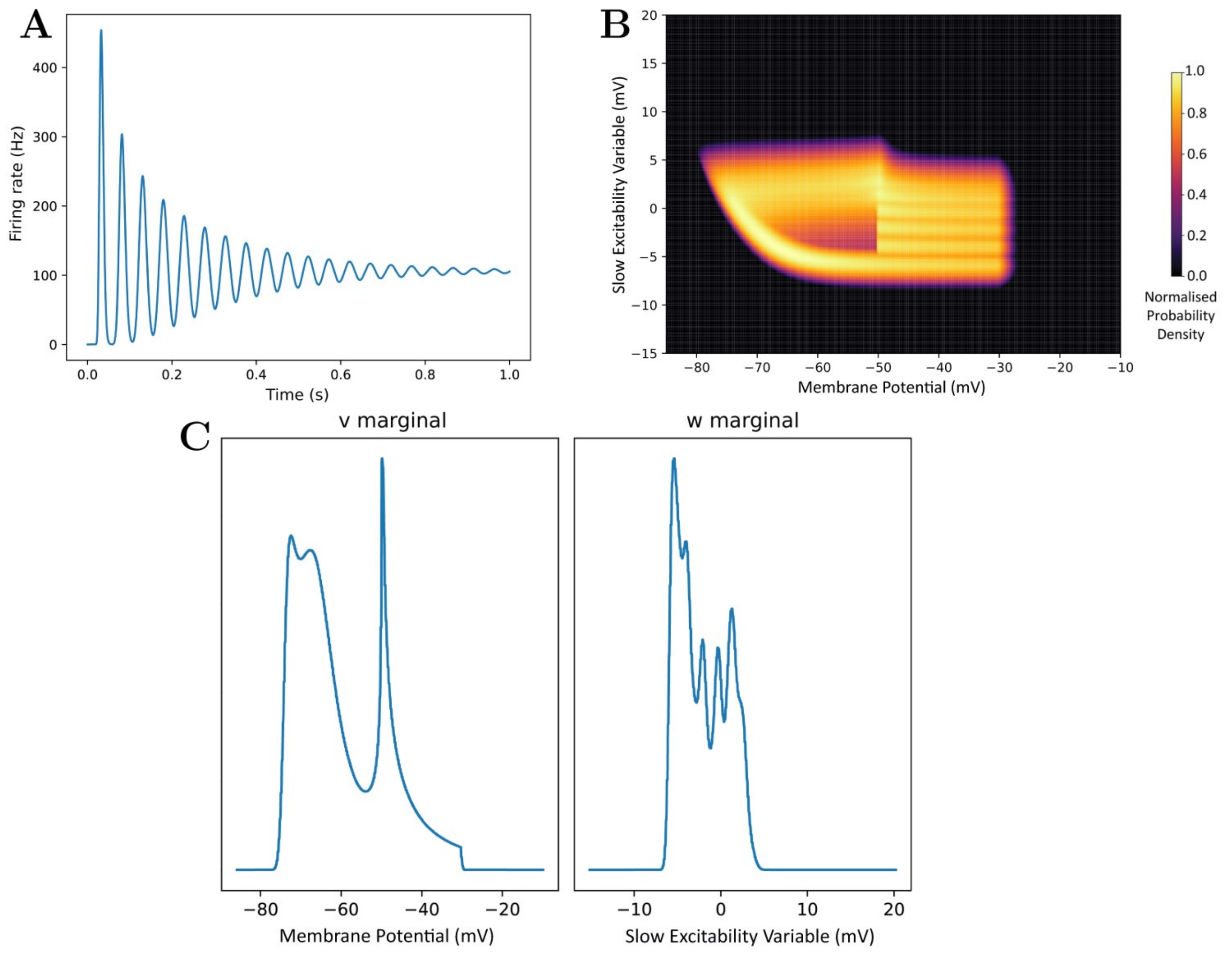


Figure 2.11: (A) The average firing rate of a population produced by calling the **rate** command. (B) A density plot (normalised to the maximum density value) of the population produced by calling the **plot-density** command. (C) The marginal density plots produced by calling the **plot-marginals** command.

parameters identifying the required node name and simulation time.

Listing 2.30: Plot the probability density of population POP1 at time 0.42s in the CLI.

```
> plot-density POP1 0.42
```

This command renders the mesh or grid and its population density at the given simulation time. When reading the simulation time parameter in the command, MIIND expects the time to be an integer multiple of the time step and to be expressed up to its least significant figure (for example, 0.1 instead of 0.10). Again, this command can be run during a simulation providing the time has been simulated. An example of a density plot is shown in Fig. 2.11B.

Similar to **plot-density**, **plot-marginals** can be used to display the marginal densities of a given population at a given time. Both marginals are plotted next to each other. The details of how marginal densities are calculated are explained in the supplementary material section B.5. Fig. 2.11C shows an example of a marginal density plot.

Listing 2.31: Plot the marginal distributions of population POP1 at time 0.42s in the CLI.

```
> plot-marginals POP1 0.42
```

2.8.1 Generate a Density Movie

If, in the XML file <*Recording*> section, the <*Display*> element is added for a given population, the output directory will be populated with still images of density plots at each time step. Once the simulation is complete, calling **generate-density-movie** in the CLI will produce an MP4 movie file made from the still images. The parameters are the node name followed by the size of the square video frame in pixels. The third parameter is the desired time to display each image (every time step of the simulation) in seconds. If the video should be the same length as the simulation time, then this parameter should match the time step of the simulation. By changing the value, the video time can be altered. For example, if the parameter is set to 0.01 for a simulation with time step 0.001, then the video length will be 10 times the length of the simulation. Finally, a name for the video file must be given.

Listing 2.32: Generate a movie from the display images of population POP1 with a size of 512 pixels at a simulation replay time step of 0.1s.

```
> generate-density-movie POP1 512 0.1 pop1_mov
```

The movie file will be created in the working directory of the simulation. A movie of the marginal density plots can also be created using the **generate-marginal-movie** command which takes the same parameters. As each marginal plot must be generated from the density output, this takes a considerably longer time than for the density movie.

Listing 2.33: Generate a marginals movie from the density files of population POP1 with a size of 512 pixels at a simulation replay time step of 0.1s.

```
> generate-marginal-movie POP1 512 0.1 pop1_marginal_mov
```

2.9 Description of MIIND's architecture and functionality

The main architectural concerns in MIIND relate to the two C++ libraries, MPILib and TwoDLib. MPILib is responsible for instantiating and running the simulation. TwoDLib contains the CPU implementations of the grid and mesh algorithms. It is also responsible for generating transition matrices. Of the remaining libraries, GeomLib contains a population density technique implementation of neuron models with one time-dependent variable, although it is also possible and indeed preferable to use the TwoDLib code for one-dimensional models. EPFLLib and NumtoolsLib contain helper classes and type definitions. Fig. 2.12 shows a reduced UML diagram of the MIIND C++ architecture. The aim of this section is to give a brief overview of the C++ MIIND code as a starting point for developers. The CUDA implementation of MIIND is similar in structure to the CPU solution and is available in the CudaTwoDLib and MiindLib libraries. A description of the differences is given in section B.3 of the supplementary material.

2.9.1 MPILib

The MPINetwork class in MPILib represents a simulation as a whole and is instantiated in the *init* function of the SimulationParserCPU class which is a specialisation of MiindTvbModelAbstract. *init* is called from the Python module and, as the name suggests, MiindTvbModelAbstract was originally written with the aim of Python integration into TVB. MPINetwork exposes member functions for building a network of nodes where each node is an instance of a neuron population which can be connected together so that the output activity from one population is input to another. The class also contains all of the simulation parameters such as the simulation length and time step. Finally, the MPINetwork class exposes a function to run the simulation in its entirety or take a single evolve step for use in an external control loop.

Each node in the population network is represented by an instance of the MPINode class. A node has a name and an ID which is used to uniquely identify it in the simulation. A node also contains an implementation of AlgorithmInterface performing the integration technique required for this population (for example, GridAlgorithm or MeshAlgorithm). The *NodeType* describes whether a population should

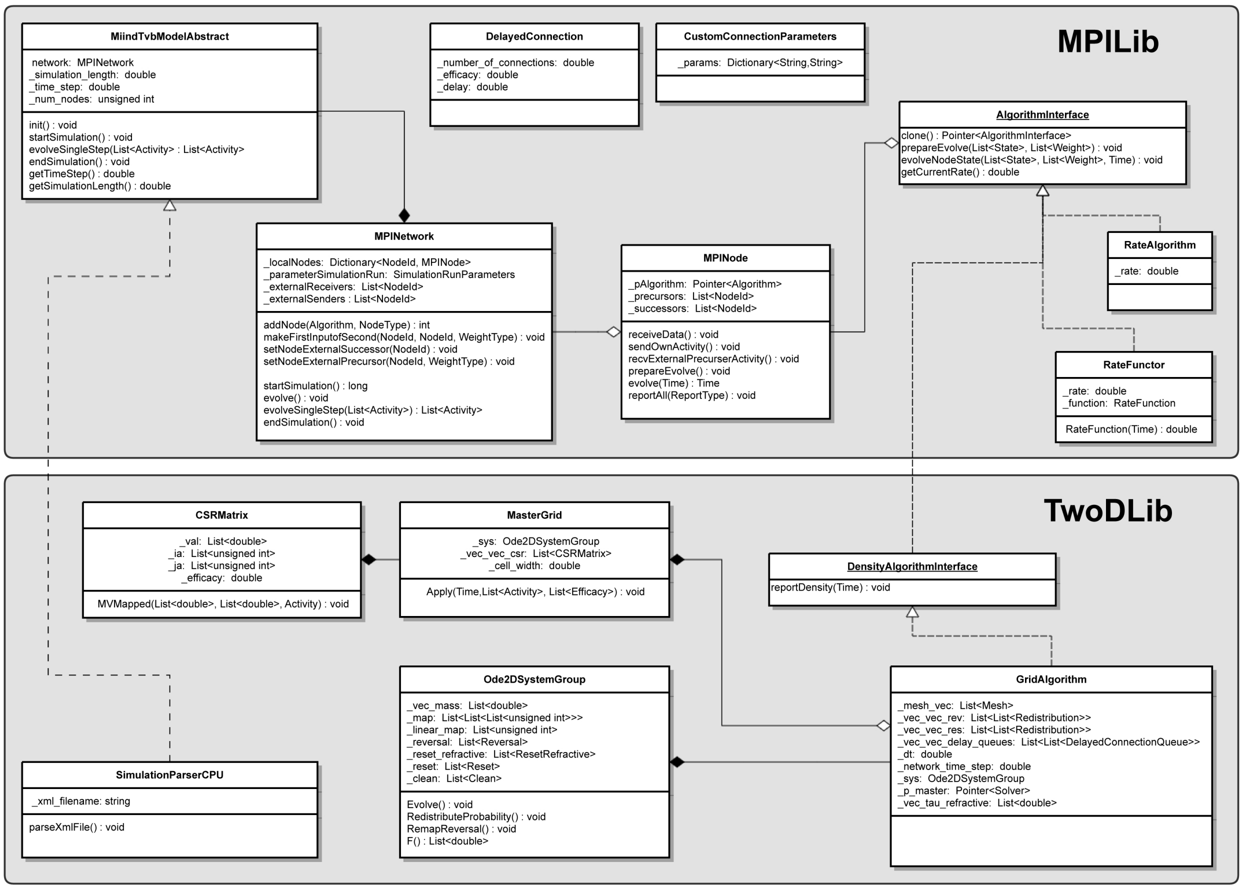

MPILib
MiindTvbModelAbstract
network: MPINetwork
_simulation_length: double
_time_step: double
_num_nodes: unsigned int
init() : void
startSimulation() : void
evolveSingleStep(List<Activity> : List<Activity>
endSimulation() : void
getTimeStep() : double
getSimulationLength() : double
DelayedConnection
_number_of_connections: double
_efficacy: double
_delay: double
CustomConnectionParameters
_params: Dictionary<String,String>
AlgorithmInterface
clone() : Pointer<AlgorithmInterface>
prepareEvolve(List<State>, List<Weight>) : void
evolveNodeState(List<State>, List<Weight>, Time) : void
getCurrentRate() : double
MPINetwork
_localNodes: Dictionary<NodeId, MPINode>
_parameterSimulationRun: SimulationRunParameters
_externalReceivers: List<NodeId>
_externalSenders : List<NodeId>
addNode(Algorithm, NodeType) : int
makeFirstInputofSecond(NodeId, NodeId, WeightType) : void
setNodeExternalSuccessor(NodeId) : void
setNodeExternalPrecursor(NodeId, WeightType) : void
startSimulation() : long
evolve() : void
evolveSingleStep(List<Activity>) : List<Activity>
endSimulation() : void
MPINode
_pAlgorithm: Pointer<Algorithm>
_precursors: List<NodeId>
_successors: List<NodeId>
receiveData() : void
sendOwnActivity() : void
recvExternalPrecurserActivity() : void
prepareEvolve() : void
evolve(Time) : Time
reportAll(ReportType) : void
RateAlgorithm
_rate: double
RateFunctor
_rate: double
_function: RateFunction
RateFunction(Time) : double
TwoDLib
CSRMatrix
_val: List<double>
_ia: List<unsigned int>
_ja: List<unsigned int>
_efficacy: double
MVMapped(List<double>, List<double>, Activity) : void
MasterGrid
_sys: Ode2DSystemGroup
_vec_vec_csr: List<CSRMatrix>
_cell_width: double
Apply(Time,List<Activity>, List<Efficacy>) : void
DensityAlgorithmInterface
reportDensity(Time) : void
Ode2DSystemGroup
_vec_mass: List<double>
_map: List<List<List<unsigned int>>>
_linear_map: List<unsigned int>
_reversal: List<Reversal>
_reset_refractive: List<ResetRefractive>
_reset: List<Reset>
_clean: List<Clean>
Evolve() : void
RedistributeProbability() : void
RemapReversal() : void
F() : List<double>
GridAlgorithm
_mesh_vec: List<Mesh>
_vec_vec_rev: List<List<Redistribution>>
_vec_vec_res: List<List<Redistribution>>
_vec_vec_delay_queues: List<List<DelayedConnectionQueue>>
_dt: double
_network_time_step: double
_sys: Ode2DSystemGroup
_p_master: Pointer<Solver>
_vec_tau_refractive: List<double>
SimulationParserCPU
_xml_filename: string
parseXmlFile() : void

Figure 2.12: A minimal UML diagram of MIIND. The two major libraries, MPILib and TwoDLib, are represented.

be thought of as excitatory or inhibitory. As discussed earlier, MIIND performs a validation check that the synaptic efficacy from a node is positive or negative respectively (or neutral). During each iteration, each node is responsible for consolidating the activity of all input connections, calling the integration step in the AlgorithmInterface implementation, and reporting the density and output activity (the average firing rate or membrane potential).

In MPILib, a number of implementations of AlgorithmInterface are defined which can be instantiated in a node. Implementations of AlgorithmInterface are responsible for the lion's share of the computation in MIIND as this is where the integration of the model is performed. The interface is extremely simple, providing a function to set parameters, an optional function for a preamble before each iteration, and the *evolveNodeState* function to be called every time step. GridAlgorithm and MeshAlgorithm are implementations of this interface defined in TwoDLib. MPILib and GeomLib hold the implementations of the remaining algorithms available to the user which were discussed in section 2.5. Finally, the weight types, DelayedConnection and CustomConnectionParameters are also defined in MPILib. All classes are C++ templates which take the weight type as a parameter to avoid code duplication and to enforce that only algorithms with the same weight type can be used together.

2.9.2 TwoDLib

As with the population models in MPILib and GeomLib, GridAlgorithm and MeshAlgorithm are implementations of the AlgorithmInterface. We will focus here on the grid algorithm implementation although the mesh algorithm uses the same structures or specialisations of those structures to perform similar tasks as set out in section 2.4. GridAlgorithm is supported by two important classes. **Ode2DSystem** transfers probability mass according to the reset mapping of the *.model* file and calculates the average firing rate of the population. In MeshAlgorithm, Ode2DSystem also performs the pointer update for shifting probability mass down the strips of the mesh. **MasterGrid** is responsible for solving the Poisson master equation using a transition matrix calculated at simulation time based on the desired efficacy and grid cell size. For each iteration, the function *evolveNodeState* is called which performs the main steps of the population density algorithm.

First, in GridAlgorithm, the deterministic dynamics are solved by applying the pre-generated transition matrix once. The second step is a call to *Ode2DSystem.RedistributeProbability()* to perform any reset mappings for probability mass which appeared in the threshold cells last iteration. This step is useful for neuron models, such as leaky integrate-and-fire, which contain an instruction to reset one or more variables to a different value upon reaching a threshold.

The third step calls on the MasterGrid class to solve the master equation for the incoming Poisson spike rates from every incident node. MasterGrid begins with the current state of the probability mass distribution across the grid, that is, the probability mass values of each cell in the grid. As described in section 2.3, every cell has the same relative transition of probability mass due to a single incoming spike. For the whole grid, this single transition is duplicated into a transition matrix which can be applied to the full probability mass vector. Because there are at most two cells into which probability mass is transferred, this matrix is extremely sparse and can be stored efficiently in a compressed sparse row (CSR) matrix. In the mesh algorithm, this matrix is loaded from the *.mat* file.

MeshAlgorithm requires a fourth step to transfer probability mass from the ends of strips to stationary cells subject to a reversal mapping generated during the pre-processing phase. This is discussed in the supplementary material section B.6.

Finally, SimulationParserCPU is an extension of the MiindTvbModelAbstract class used to parse the simulation XML file and instantiate an MPINetwork object with the appropriate nodes and connections. Its extensions of the functions declared in MiindTvbModelAbstract are exposed to the Python module to be called from a Python script.

2.10 Discussion

MIIND fulfils a need for insight into neural behaviour at mesoscopic scales.

The MIIND population density technique allows researchers to simulate population-level behaviour by defining the behaviour of the underlying neurons. This is in contrast to many rate-based models which describe the population behaviour directly. An example of how population behaviour can differ from the underlying neuron model can be seen in the behaviour of a population of bursting neurons such as the Izhikevich simple model. A single Izhikevich neuron with a constant input current or input spike rate oscillates between a bursting period of repeated firing and a quiescent period of no firing. The average behaviour of a population of Izhikevich neurons is different. Initially, all neurons are synchronised, they burst and quiesce at the same time producing an oscillatory pattern of average firing rate in the population. However, due to the random nature of Poisson input spikes, the neurons de-synchronise over time and the average firing rate of the whole population damps to a constant value because only a subset of neurons is bursting at any one time. Fig. 2.11A shows the damping of the output firing rate oscillations and the 'desynchronised' density of a population of Izhikevich simple neurons.

TVB Integration

The Virtual Brain (Sanz Leon et al., 2013) and MIIND are both systems which facilitate the development of neural mass or mean-field population models with explicit descriptions of how multiple populations are connected. Using these systems, the complex dynamics arising from the interaction of populations can be studied. TVB provides a framework to describe a network of nodes (the connectivity) which, while it can be abstract, generally represents regions of the human or primate brain. Connections between nodes represent white matter tracts which transfer signals from one node to the next based on length and propagation speed. TVB also allows the description of "coupling" functions which modulate these signals as they pass from one node to another. Typically, the number of nodes is in the order of 100 or so. However, TVB also allows for the definition of a "surface" which can be associated with 10s of thousands of nodes to simulate output from common medical recording techniques such as EEG and BOLD fMRI. TVB has impressive clinical relevance as well as supporting more theoretical neuroscience research. Users can build simulations using the graphical user interface or directly using the Python source code.

While MIIND and TVB have many functional similarities, both have differing strengths with respect to the underlying simulation techniques and surrounding infrastructure. It was therefore clear that integrating the smaller system, MIIND, into the more developed infrastructure of TVB might yield benefits from both.

Although it is possible to model delayed connections and synaptic dynamics between populations in MIIND, TVB provides a comprehensive method of defining such structures and behaviours through the connectivity network and coupling functions. Some users of MIIND may find it useful and appropriate to house their simulations in such a structure.

TVB uses a number of model classes to describe the behaviour of the nodes in a network. When the simulation is run, an instantiation of a specified model class takes the signals which have passed through the network to arrive at each node and integrates forward by one time step (depending on the integration method). In order to use MIIND nodes in TVB, a specialised model class was created to import the MIIND Python library, instantiate it, then make a call to *evolveSingleStep()* in place of the integration function. The inputs and outputs of *evolveSingleStep()* are treated by TVB as any other model. As the MIIND Python library takes a simulation file name as a parameter to its *init* function, a single additional model class is all that is required to expose any MIIND simulation to TVB. Fig. 2.13 shows the results from a simulation of the TVB default whole-brain connectivity with populations of Izhikevich simple neurons in MIIND. The script and simulation files are available in the *examples/miind_tvb* directory of the MIIND repository. Both TVB and MIIND must be installed to successfully run the example.

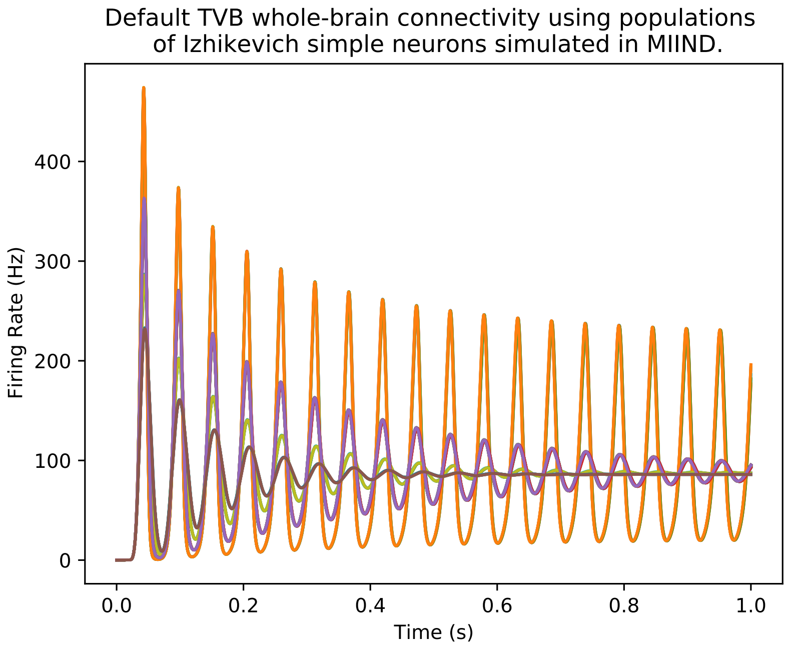


Figure 2.13: The firing rates of 76 nodes from the default TVB connectivity simulation. Each node is a population of Izhikevich simple neurons simulated using MIIND. The majority of nodes produce oscillations which decay to a constant average firing rate. However, a subset of nodes remains in an oscillating state.

Reasoning about probability density instead of populations of individual neurons simplifies output analysis.

The output firing rate or membrane potential of a MIIND population which uses the mesh algorithm or grid algorithm is devoid of any variation which you would see from a population of individual neurons. This is because the effect of Poisson-generated input spike trains is applied to a probability density function, effectively an infinite population of neurons. Spike train inputs to a finite population of neurons produces variation in how individual neurons move through state space resulting in noisy output rates at the population level. While this can be mitigated using a larger number of neurons, the use of smoothing techniques, or curve fitting, MIIND requires none of these methods to produce an output which is immediately clear to interpret. For example, MIIND was used to build and simulate a spinal circuit model using populations of integrate-and-fire neurons (York et al., 2021). The average firing rates of the populations were used to compare patterns of activity with results from an EMG experiment. As the patterns to be observed were on the order of seconds, there was no need to capture faster variation in activity from the simulation and indeed, a direct simulation would have produced output which may have obscured these patterns.

MIIND has also been used to simulate central pattern generator models which rely on mutually inhibiting populations of bursting neurons. The interaction of the two populations significantly influences their sub-threshold dynamics. In particular, it can be difficult to identify the dynamics responsible for the swapping of states from bursting to quiescent (escape or release). Observing the changing probability density function during the simulation makes it very clear how the two populations are behaving.

Handling Noise

A major benefit of MIIND's population density technique is the ability to observe the effect of noise on a population and to manipulate noise in an intuitive way. For a given simulation, the Poisson distributed input to a population causes a spread of probability mass across the state space as some neurons receive many spikes, and some receive fewer. It is explained by de Kamps, 2013 how the Poisson input causes a mean increase in membrane potential equal to the product of the post synaptic efficacy, h, and the average input rate, ν. It causes a variance equal to νh^2. h and ν can therefore be set such that the mean remains the same but the variance changes to observe the effect of noise on the population.

Another simple way to increase the variance of the population is to introduce two additional inputs with equal rates and opposite post-synaptic efficacies. Again, the mean increase caused by the input remains unchanged but the variance can be increased significantly and this requires only a small change to the XML simulation file.

A model agnostic system at the population level makes prototyping quick and intuitive.

Because MIIND provides insight into how a neuron model produces behaviour at the population level, it is beneficial that the grid algorithm enables the user to quickly reproduce the *.model* and *.tmat* files if the underlying neuron model needs to be changed. An example of this can be observed in a half-centre oscillator made of a pair of mutually inhibiting populations of bursting neurons. The frequency of oscillation can be made dependent or independent of the input spike rate by including a limit on the slow excitability variable of the underlying neuron model. To make this change, the user can alter the neuron model then rebuild the *.model* and *.tmat* file and no change to the population level network is required.

DiPDE

DiPDE (DiPDE, 2015; Iyer et al., 2013) is an alternative implementation of the population density technique for one dimensional neuron models. It does not employ the "mesh" discretisation method used in the MIIND mesh algorithm and has primarily been used with populations of leaky integrate-and-fire neurons. DiPDE can be used to simulate the Potjans-Diesmann microcircuit model (Cain et al., 2016) which shows good agreement with MIIND (Fig. 2.5). MIIND is a much larger application than DiPDE because it allows users to design their own underlying neuron models for each population using either the mesh or grid algorithms.

Future Work

A limitation of the MIIND population density technique is that a maximum of two time-dependent variables can be used to describe the underlying neuron model of each population. In the mesh algorithm, for higher dimensions, mesh building would need to be automated but this is not a trivial problem to solve. The grid algorithm, however, is entirely automated and work has been done to extend MIIND for 3D neuron models. Fig. 2.14 shows the 3D density plot of a population of Hindmarsh-Rose neurons in MIIND. The technique used to generate the 2D transition matrices outlined in section 2.3 extends to N dimensions so there is theoretically no limit to the dimensionality of the underlying neuron model in the grid algorithm. However, both the grid algorithm and mesh algorithm suffer from "the curse of dimensionality" such that with each additional variable, the number of cells to cover the state space increases to the point where the memory and processing requirements are too high. Luckily, a great number of neuron behaviours can be captured with only two or three time-dependent variables with appropriate approximations.

Large networks can be built up quickly in MIIND. To add a node to a simulation file requires just a single line. Integrating the node into the rest of the network with requisite connections is equally

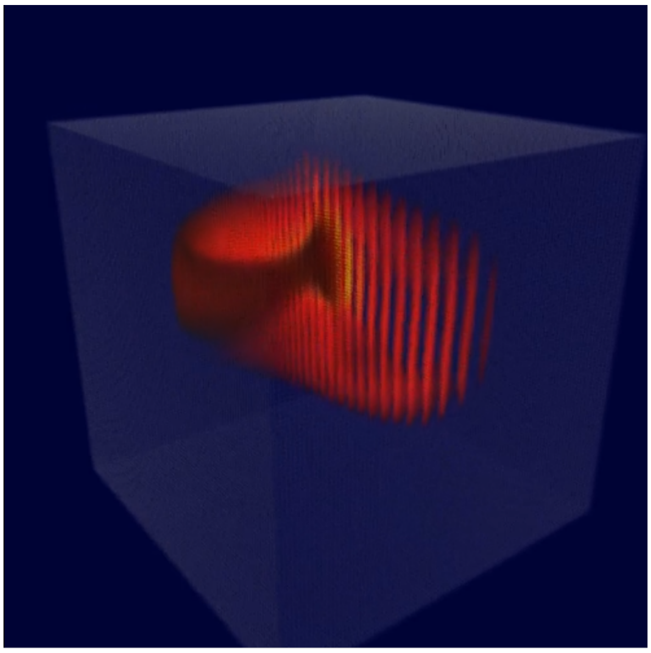


Figure 2.14: (A) A density plot of a population of Hindmarsh-Rose neurons. The density is contained in a three-dimensional volume such that each axis represents one of the time-dependent variables of the model. The volume has been rendered from a rotated and elevated position to more easily visualise the density.

convenient. As mentioned, the Potjans-Diesmann model has been implemented as a single cortical column but this is by no means the limit of the size of the network which can be built. It is feasible that a patch of cortex made of perhaps hundreds of cortical columns can be simulated efficiently in MIIND. The benefit of such a network would be to demonstrate how cortical columns interact together under different connectivity regimes and inputs as well as providing the ability to quickly and easily "swap out" the underlying neuron model of each population. Typically, LIF is used but adaptive integrate-and-fire would be a closer approximation to pyramidal neurons in cortex.

Conclusion

We have presented the mesh and grid algorithms, MIIND's population density techniques for simulating populations of neurons and given a full account of the software features available to users. While the mesh algorithm was developed some time ago, the grid algorithm which was added to MIIND recently has precipitated a more accessible, user-friendly software package. We hope that the explanations given here along with a lower technical barrier to entry will encourage researchers to make use of the tool.

Data Availability Statement

The MIIND source code and installation packages are available as a github repository at `https://github.com/dekamps/miind`.

MIIND can be installed for use in Python using “pip install miind” on many Linux, MacOS, and Windows machines with python versions >= 3.6.

Documentation is available at `https://miind.readthedocs.io/`.

References

Amit, Daniel J and Brunel, Nicolas (1997). "Model of global spontaneous activity and local structured activity during delay periods in the cerebral cortex." In: *Cerebral cortex (New York, NY: 1991)* 7.3, pp. 237–252.

Bower, James M and Beeman, David (2012). *The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System.* Springer Science & Business Media New York.

Brette, Romain and Gerstner, Wulfram (2005). "Adaptive exponential integrate-and-fire model as an effective description of neuronal activity". In: *Journal of neurophysiology* 94.5, pp. 3637–3642.

Brunel, Nicolas and Hakim, Vincent (1999). "Fast global oscillations in networks of integrate-and-fire neurons with low firing rates". In: *Neural computation* 11.7, pp. 1621–1671.

Cain, Nicholas, Iyer, Ramakrishnan, Koch, Christof, and Mihalas, Stefan (2016). "The computational properties of a simplified cortical column model". In: *PLoS computational biology* 12.9, e1005045.

Carlu, Mallory, Chehab, Omar, Dalla Porta, Leonardo, Depannemaecker, Damien, Héricé, Charlotte, Jedynak, Maciej, Köksal Ersöz, E, Muratore, Paolo, Souihel, Selma, Capone, Cristiano, et al. (2020). "A mean-field approach to the dynamics of networks of complex neurons, from nonlinear Integrate-and-Fire to Hodgkin–Huxley models". In: *Journal of neurophysiology* 123.3, pp. 1042–1051.

D'Angelo, Egidio, Antonietti, Alberto, Casali, Stefano, Casellato, Claudia, Garrido, Jesus A, Luque, Niceto Rafael, Mapelli, Lisa, Masoli, Stefano, Pedrocchi, Alessandra, Prestori, Francesca, et al. (2016). "Modeling the cerebellar microcircuit: new strategies for a long-standing issue". In: *Frontiers in cellular neuroscience* 10, p. 176.

de Kamps, Marc (2003). "A simple and stable numerical solution for the population density equation". In: *Neural computation* 15.9, pp. 2129–2146.

– (2013). "A generic approach to solving jump diffusion equations with applications to neural populations". In: *arXiv preprint arXiv:1309.1654*. URL: `https://arxiv.org/abs/1309.1654v2`.

de Kamps, Marc, Baier, V, Drever, J, Dietz, M, Mösenlechner, L, and Van der Velde, F (2008). "The state of MIIND". In: *Neural Networks* 21.8, pp. 1164–1181.

de Kamps, Marc, Lepperød, Mikkel, and Lai, Yi Ming (2019). “Computational geometry for modeling neural populations: From visualization to simulation”. In: *PLoS computational biology* 15.3, e1006729.

DiPDE (2015). *Website: © 2015 Allen Institute for Brain Science. DiPDE Simulator [Internet]. Available from: https://github.com/AllenInstitute/dipde.*

El Boustani, Sami and Destexhe, Alain (2009). “A master equation formalism for macroscopic modeling of asynchronous irregular activity states”. In: *Neural computation* 21.1, pp. 46–100.

FitzHugh, Richard (1961). “Impulses and physiological states in theoretical models of nerve membrane”. In: *Biophysical journal* 1.6, p. 445.

Fourcaud-Trocmé, Nicolas, Hansel, David, Van Vreeswijk, Carl, and Brunel, Nicolas (2003). “How spike generation mechanisms determine the neuronal response to fluctuating inputs”. In: *Journal of Neuroscience* 23.37, pp. 11628–11640.

Furber, Steve B., Galluppi, Francesco, Temple, Steve, and Plana, Luis A. (2014). “The SpiNNaker project”. In: *IEEE. Proceedings* 102.5, pp. 652–665. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2304638.

Gerstner, Wulfram (1998). *Spiking neurons.* Tech. rep. MIT-press.

Gewaltig, Marc-Oliver and Diesmann, Markus (2007). “NEST (NEural Simulation Tool)”. In: *Scholarpedia* 2.4, p. 1430.

Hindmarsh, James L and Rose, RM (1984). “A model of neuronal bursting using three coupled first order differential equations”. In: *Proceedings of the Royal society of London. Series B. Biological sciences* 221.1222, pp. 87–102.

Hines, Michael L and Carnevale, Nicholas T (2001). “NEURON: a tool for neuroscientists”. In: *The neuroscientist* 7.2, pp. 123–135.

Iyer, Ramakrishnan, Menon, Vilas, Buice, Michael, Koch, Christof, and Mihalas, Stefan (2013). “The influence of synaptic weight distribution on neuronal population dynamics”. In: *PLoS computational biology* 9.10, e1003248.

Izhikevich, Eugene M (2003). “Simple model of spiking neurons”. In: *IEEE Transactions on neural networks* 14.6, pp. 1569–1572.

– (2007). *Dynamical systems in neuroscience: The Geometry of Excitability and Bursting.* Cambridge, MA: MIT press.

Jirsa, Viktor K, Stacey, William C, Quilichini, Pascale P, Ivanov, Anton I, and Bernard, Christophe (2014). “On the nature of seizure dynamics”. In: *Brain* 137.8, pp. 2210–2230.

Knight, Bruce W (1972). “Dynamics of encoding in a population of neurons”. In: *The Journal of general physiology* 59.6, pp. 734–766.

Knight, Bruce W, Manin, Dimitri, and Sirovich, Lawrence (1996). “Dynamical models of interacting neuron populations in visual cortex”. In: *Robot Cybern* 54, pp. 4–8.

Lai, Yi Ming and de Kamps, Marc (2017). “Population density equations for stochastic processes with memory kernels”. In: *Physical Review E* 95.6, p. 062125.

Mattia, Maurizio and Del Giudice, Paolo (2002). “Population dynamics of interacting spiking neurons”. In: *Physical Review E* 66.5, p. 051917.

– (2004). “Finite-size dynamics of inhibitory and excitatory interacting spiking neurons”. In: *Physical Review E* 70.5, p. 052903.

Montbrió, Ernest, Pazó, Diego, and Roxin, Alex (2015). “Macroscopic description for networks of spiking neurons”. In: *Physical Review X* 5.2, p. 021028.

Nagumo, Jinichi, Arimoto, Suguru, and Yoshizawa, Shuji (1962). “An active pulse transmission line simulating nerve axon”. In: *Proceedings of the IRE* 50.10, pp. 2061–2070.

Nykamp, Duane Q and Tranchina, Daniel (2000). “A population density approach that facilitates large-scale modeling of neural networks: Analysis and an application to orientation tuning”. In: *Journal of computational neuroscience* 8.1, pp. 19–50. DOI: 10.1023/A:1008912914816. URL: https://doi.org/10.1023/A:1008912914816.

Omurtag, Ahmet, Knight, Bruce W, and Sirovich, Lawrence (2000). "On the simulation of large populations of neurons". In: *Journal of computational neuroscience* 8.1, pp. 51–63.

Osborne, Hugh and de Kamps, Marc (2021). *MIIND Documentation [Internet]. Available from: https://miind.readthedocs.io.*

Potjans, Tobias C and Diesmann, Markus (2014). "The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model". In: *Cerebral cortex* 24.3, pp. 785–806.

Proix, Timothée, Bartolomei, Fabrice, Guye, Maxime, and Jirsa, Viktor K (2017). "Individual brain structure and modelling predict seizure propagation". In: *Brain* 140.3, pp. 641–654.

Sanz Leon, Paula, Knock, Stuart A, Woodman, M Marmaduke, Domide, Lia, Mersmann, Jochen, McIntosh, Anthony R, and Jirsa, Viktor (2013). "The Virtual Brain: a simulator of primate brain network dynamics". In: *Frontiers in neuroinformatics* 7, p. 10.

Traub, Roger D, Contreras, Diego, Cunningham, Mark O, Murray, Hilary, LeBeau, Fiona EN, Roopun, Anita, Bibbig, Andrea, Wilent, W Bryan, Higley, Michael J, and Whittington, Miles A (2005). "Single-column thalamocortical network model exhibiting gamma oscillations, sleep spindles, and epileptogenic bursts". In: *Journal of neurophysiology* 93.4, pp. 2194–2232.

Tsodyks, Misha V and Markram, Henry (1997). "The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability". In: *Proceedings of the national academy of sciences* 94.2, pp. 719–723.

Uhlenbeck, George E and Ornstein, Leonard S (1930). "On the theory of the Brownian motion". In: *Physical review* 36.5, p. 823.

Wilson, HR and Cowan, JD (1972). "Excitatory and inhibitory interactions in localized populations of model neurons". In: *Biophysical journal* 12.1, pp. 1–24.

Wilson, Matthew A, Bhalla, Upinder S, Uhley, John D, and Bower, James M (1988). "GENESIS: a system for simulating neural networks". In: *Proceedings of the 1st International Conference on Neural Information Processing Systems*. Morgan Kaufmann, pp. 485–492.

York, Gareth, Osborne, Hugh, Sriya, Piyanee, Astill, Sarah, de Kamps, Marc, and Chakrabarty, Samit (2021). "Muscles Recruited During an Isometric Knee Extension Task is Defined by Proprioceptive Feedback". In: *BioRxiv*. DOI: 10.1101/802736. URL: https://www.biorxiv.org/content/early/2021/04/29/802736.

Chapter 3

A Numerical Population Density Technique for N-Dimensional Neuron Models (Paper B)

3.1 Abstract

Population density techniques can be used to simulate the behaviour of a population of neurons which adhere to a common underlying neuron model. They have previously been used for analysing models of orientation tuning and decision-making tasks. They produce a fully deterministic solution to neural simulations which often involve a non-deterministic or noise component. Until now, numerical population density techniques have been limited to only one- and two-dimensional models. For the first time, we demonstrate a method to take an N-dimensional underlying neuron model and simulate the behaviour of a population. The technique enables so-called graceful degradation of the dynamics allowing a balance between accuracy and simulation speed while maintaining important behavioural features such as rate curves and bifurcations. It is an extension of the numerical population density technique implemented in the MIIND software framework that simulates networks of populations of neurons. Here, we describe the extension to N dimensions and simulate populations of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances then demonstrate the effect of degrading the accuracy of the solution. We also simulate two separate populations in an E-I configuration to demonstrate the technique's ability to capture complex behaviours of interacting populations. Finally, we simulate a population of four-dimensional Hodgkin-Huxley neurons under the influence of noise. Though the MIIND software has been used only for neural modelling up to this point, the technique can be used to simulate the behaviour of a population of agents adhering to any

system of ordinary differential equations under the influence of shot noise. MIIND has been modified to render a visualisation of any three of an N-dimensional state space of a population which encourages fast model prototyping and debugging and could prove a useful educational tool for understanding dynamical systems.

3.2 Introduction

A common and intuitive method for simulating the behaviour of a population of neurons is to directly simulate each individual neuron and aggregate the results (Gewaltig and Diesmann, 2007; Yavuz et al., 2016; J. C. Knight et al., 2021). At this level of granularity, the population can be heterogeneous in terms of the neuron model used, parameter values, and connections. The state of each neuron, which may consist of one or many more time or spatially dependent variables, is then integrated forward in time. The benefit of this method of simulation is that it provides a great deal of control over the simulated neurons with the fewest approximations. If required, the state history of each neuron can be inspected. However, this degree of detail can produce results that are overly verbose making it difficult to explain observations. While this can be mitigated by carefully limiting the degrees of freedom (for example, keeping all neurons in the population homogeneous, using point neuron models, or having a well-defined connection heuristic), other simulation methods exist that have such assumptions built-in and provide additional benefits like increased computation speed, lower memory requirements, or improved ways to present and interpret the data. For example, so-called neural mass models (Wilson and Cowan, 1972; Jansen and Rit, 1995) eschew the behaviour of the individual neurons in a population in favour of a direct definition of the average behaviour. These methods are computationally cheap and can be based on empirical measurements but they lack a direct link to the microscopic behaviour of the constituent neurons which limits a generalisation to populations of different neuron types.

Population density techniques (PDTs) approximate population-level behaviours based on a model definition of the constituent neurons. Most PDTs assume all neurons are homogeneous and unconnected within a discrete population. All neurons are considered point-neurons and adhere to a single neuron model which is made up of one or more variables that describe the state of the neuron at a given time. The state-space of the model, as shown in Fig. 3.1, contains all possible states that a neuron in the population could take. For a population of neurons, PDTs frequently define a probability density function or the related probability mass function across the state space which gives the probability of finding a neuron from the population with a given state. PDTs are not concerned with the individual neurons but instead calculate the change to the probability mass function which is governed by two processes: the deterministic dynamics defined by the underlying neuron model, and a non-deterministic noise process representing random incoming spike events.

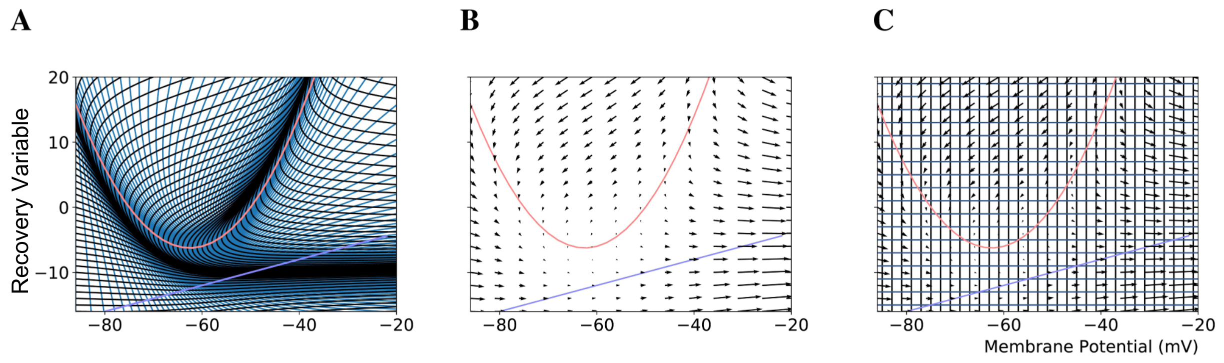


Figure 3.1: (A) The mesh used in MIIND to simulate a population of Izhikevich neurons. The quadratic red curve and blue line are the nullclines where the rate of change of the membrane potential and recovery variable respectively are zero. The strips, made up of quadrilateral cells are formed by the characteristic curves of the Izhikevich model for a given parameter set. (B) A vector field for the same model showing the direction of movement of probability mass around the state space. (C) The state-space discretised into a regular grid. The parameters and definition of the Izhikevich model are not given here as it is only required to demonstrate the mesh and grid discretisation. As in the original derivation of the model, the recovery variable has no units.

Methods for solving the deterministic dynamics of a system of ordinary differential equations under the influence of a non-deterministic noise process have been used right back to early studies of Brownian motion. Then in theoretical neuroscience Johannesma, 1969 and B. W. Knight, 1972 among others used similar techniques to give a formal definition of the effect of stochastic spiking events on a neuron by defining a probability density function of possible somatic membrane potentials. Most often, these involved the assumption of infinitesimal changes in state due to the incoming events, also known as the diffusion approximation. Omurtag et al., 2000 applied the method to a population of unconnected homogeneous neurons. They separated the deterministic dynamics of a common underlying neuron model from the incoming spike train generated by a Poisson process. Originally, the motivation for their work was to more efficiently approximate the behaviour of collections of neurons in the visual cortex. Work by Sirovich et al., 1996 showed that there is a lot of redundancy in optical processing in the macaque visual cortex such that on the order of $O(10^4)$ functional visual characteristics or modalities are encoded by $O(10^8)$ neurons. It was, therefore, a reasonable approximation to treat a population of 10^4 neurons as a homogeneous group and investigate the interaction between populations. The technique was employed by Nykamp and Tranchina, 2000 to analyse mechanisms for orientation tuning. Bogacz et al., 2006 also used PDTs to model decision-making in a forced-choice task.

PDTs have since been extended to attend to various shortcomings of the original formulation. For example, there is often an assumption of Poisson distributed input to a population (Omurtag et al., 2000; Mattia and Del Giudice, 2002; Rangan and Cai, 2007) which in certain circumstances is not biologically realistic. Ly and Tranchina, 2009 outlined a technique to calculate the distribution of the output spike train of a population of LIF neurons with different input distributions (based on a renewal process - with a function involving the inter-spike interval). Instead of introducing a Poisson process for their noise term, they use a hazard function which defines the probability of an incoming spike given the time since the last spike. This allows them to handle more realistic input distributions such as a gamma distribution for certain situations and calculate the output firing rate. They are also able to derive the output statistics of a population like expected inter-spike interval and spike distribution. Further work has been done to develop so-called quasi-renewal processes (Naud and Gerstner, 2012) which define the probability of the next spike in terms of both the population level activity and the time since the last spike. Such approaches can simulate behaviours such as spike frequency adaptation and refractoriness but there is a weaker link to the underlying neuron model which limits the simulation of populations of neurons with dynamics that produce behaviours like bursting.

PDTs have also often been limited to low-dimensional neuron models with which to derive population-level behaviour and statistics. The conductance-based refractory density approach (Chizhov and Graham, 2007) tracks the distribution of a population of neurons according to the time since they last spiked (often referred to as their age) instead of across the state space of the neuron model. In its most elementary form, the probability density equation, given in terms of time and time since last spike, is dependent on the neuronal dynamics defined by the underlying model and a noise process. Crucially though, the conductance variables defined in the underlying model (such as the sodium gating variables of the Hodgkin-Huxley neuron model) can be approximated to their mean across all neurons with similar age. With this approximation, the dimensionality of the problem is reduced to a dependence only on the membrane potential, significantly improving the tractability of such systems. Refractory density approaches (Schwalger and Chizhov, 2019) have been extended further to approximate finite-size populations, phenomenological definitions, and bursting behaviours (Schwalger, Deger, et al., 2017; Schmutz et al., 2020; Chizhov, Campillo, et al., 2019).

Using these techniques for modelling and simulation generally requires a large amount of mathematical and theoretical work to develop a solution for a specific scenario. As we see above, each additional behaviour requires at least an extension or even reformulation of a previous approach. The numerical PDT implemented in MIIND (de Kamps et al., 2019; Osborne et al., 2021) requires only a definition of the underlying neuron model plus population and simulation parameters. The definition can be given in the form of a Python function in a similar fashion to direct simulation techniques. However, until now, the PDT has been able to simulate populations of neurons adhering to only a one- or two-dimensional model. Often, this is enough as many different neuronal behaviours can be captured with two variables, for example, the action potential of the Fitzhugh-Nagumo neuron (FitzHugh, 1961; Nagumo et al., 1962), the spike frequency adaptation of the adaptive exponential integrate-and-fire neuron (Brette and Gerstner, 2005), or the bursting behaviour of the Izhikevich neuron model (Izhikevich, 2007). Using a one-dimensional neuron model, MIIND has been employed to simulate a network of interacting populations in the spinal cord (York et al., 2022). Populations were based on the exponential integrate-and-fire neuron model and showed how a relatively simple spinal network could explain observed trends in a static leg experiment. The main benefit of using the numerical PDT in this study was to eliminate finite-size variation in the results which would have hindered the subsequent analysis. The MIIND software itself also afforded benefits such as the ability to quickly prototype population network models and to observe the population states during and after simulation. Osborne et al., 2021 have previously presented the full implementation details of MIIND including the two "flavours" of the numerical PDT, named the mesh and grid methods. The mesh method involves discretising the state space using a mesh of quadrilateral cells as shown in Fig 3.1A. The grid method was developed chiefly to improve the flexibility of the PDT to avoid building a mesh. In this method, the state space is discretised into a grid of rectangles which allows for a more automated approach. Here, we extend the grid method to greater than two-dimensional models to expand the repertoire of possible neuron types.

3.3 Materials and Methods

3.3.1 Recap of the grid method in MIIND

The MIIND algorithm for calculating the change to the probability mass function is covered in detail by de Kamps et al., 2019 and Osborne et al., 2021. However, we will cover the basic algorithm as it is relevant to the extension of the grid method to N dimensions. As a preprocessing step, the state space of the underlying neuron model is discretised such that each discrete volume of state space, or cell, is associated with a probability mass value. The probability mass is assumed to be uniformly distributed across the cell. The discretisation can take the form of a mesh as shown in Fig. 3.1A, constructed from the characteristic curves of the underlying neuron model or a regular grid which spans the state space as in Fig. 3.1C.

When generating the grid in MIIND, the user provides the resolution of the grid and the size and location in state-space within which the population is expected to remain during simulation. For each iteration of the simulation, the distribution of probability mass across the cells is updated, firstly, according to the deterministic dynamics of the underlying neuron model. For example, in the Izhikevich neuron model (Izhikevich, 2007), as shown in Fig. 3.1B, the vector field below -60mV indicates that probability mass will move slowly towards -60mV before quickly accelerating to the right. Because the underlying neuron model does not change, the proportion of probability mass transitioning from each cell according to the deterministic dynamics remains constant throughout any simulation and can therefore be precalculated and stored in a file. To generate the file, the steps illustrated in Fig. 3.2 are performed. For each cell, the aim is to calculate where probability mass will move after one time step of the simulation and how much of the mass is apportioned to each cell. First, the four vertices of the cell are translated according to a single time step of the underlying neuron model to produce a quadrilateral which is assumed to remain convex due to the small distance travelled. The quadrilateral is then split into two triangles and each triangle is then processed separately. Each triangle is tested against the axis-aligned edges of the grid. Because the lines are axis-aligned, this is a trivial test for points on either side of the line. If an intersection occurs, the new vertices are calculated to produce two polygons on either side of the line. Each polygon is triangulated and the process is recursively repeated on all sub-triangles until no more intersections occur. Once all triangles have been tested, the quadrilateral is now split into a collection of triangles which are each entirely contained within one cell of the grid. For each cell which contains one or more triangles, the total area of the triangles is calculated as a proportion of the area of the quadrilateral and this value represents the proportion of probability mass which will be transferred from the originating grid cell after one time step. It is expected that each transformed cell will only overlap with a few others in the grid so that an NxN matrix of transitions where N is the number of cells should be sparsely populated and can be stored in a file then read into memory. The transitions in the file are applied once every iteration of the simulation. This is a computationally time-efficient way to solve the deterministic dynamics.

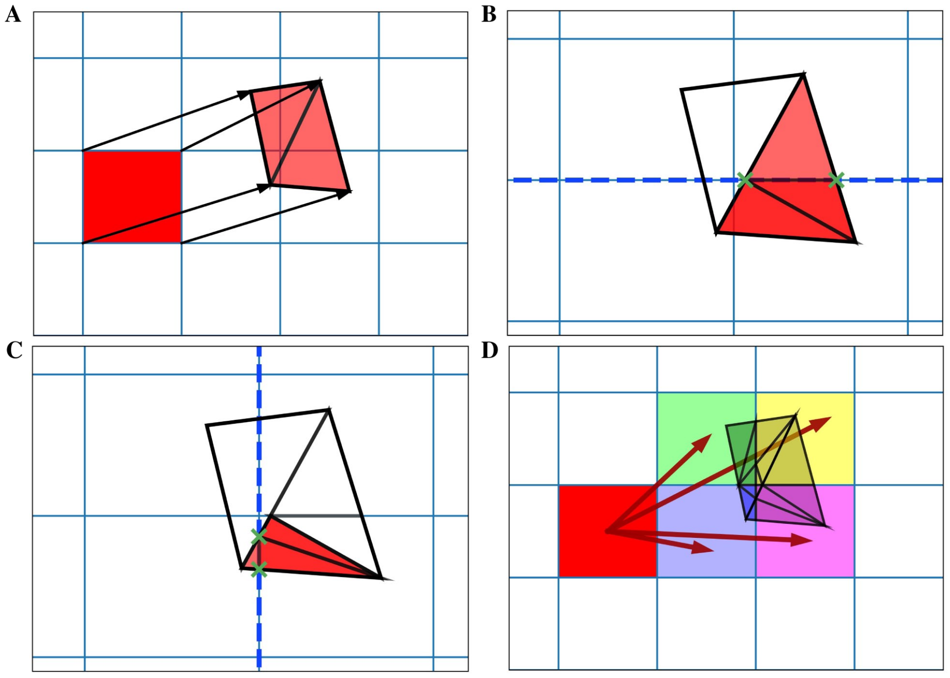


Figure 3.2: Figure showing steps for generating the transition matrix to solve the deterministic dynamics of the underlying model using a two-dimensional grid. Axes are not labelled as they represent arbitrary time-dependent variables. (A) For each grid cell (rectangle), the vertices are translated according to a single time step of the underlying neuron model and the resulting quadrilateral is triangulated. (B) Each triangle is then tested for intersection with the axis-aligned lines of the original grid. The green crosses mark the intersection points between the tested triangle and the dashed line. The resulting subsections are again triangulated. (C) The process runs recursively until no more triangulations can be made. (D) The resulting triangles each lie within only a single cell of the original grid. The area of each triangle divided by the area of the original quadrilateral gives a proportion of mass to be transferred from the grid cell to the containing cell. From these, the proportions to be transferred can be summed and the totals stored in the file.

Once the probability mass distribution has changed according to the deterministic dynamics of the underlying neuron model, the second part of the MIIND algorithm calculates the spread of mass across cells due to random (usually Poisson distributed) incoming spikes. This process is more computationally expensive than the first because the shape of the spread must be recalculated every time step by solving the Poisson master equation (de Kamps, 2006), which involves iteratively applying a different set of transitions to the probability mass function and is dependent on the incoming rate of spikes. Fig 3.3 shows how the spread of probability mass can be calculated in two dimensions based on the width of the cells and the change in state due to a single incoming spike. Calculating the transitions for solving the non-deterministic noise process benefits from the fact that all cells are the same size and regularly spaced. It is assumed that a single incoming spike will cause a neuron's state to instantaneously jump by a constant vector, J. Most often this is only in one direction instead of two. For example, many neuron models expect an instantaneous jump in membrane potential or in synaptic conductance. However, calculating the jump transition for any vector is a useful feature to have for models like the Tsodyks-Markram synapse model (Tsodyks and Markram, 1997) for which incoming spikes cause a jump in two variables at once. For a single incoming spike, all probability mass in a cell will shift up or down according to the x component of J, where x is the first variable or dimension of the model. Because all cells are the same size, this shift will result in probability mass being shared among at most two other cells which are adjacent to each other. Calculating which cells receive probability mass and in what proportion requires only knowing the width of the cells in the x dimension and the x component of J. If the J vector has a y component, where y is the second variable or dimension, the same process can be applied to each of the two new cells. The proportion of probability mass to be shared to each cell is itself shared among a further two cells for a maximum of four cells containing probability mass from the original cell. Due to the regularity of the grid, this calculation need only be made once and is applicable to every other cell. To simulate the effect of the incoming Poisson noise process on the probability mass function, the transitions are applied iteratively to each cell.

Fig. 3.3E and F show the resulting probability mass function during a simulation when both deterministic and non-deterministic processes are applied. From the function, average values across the population can be calculated as well as the average firing rate if the underlying model has a threshold-reset mechanism. In that case, after each iteration, mass that has moved into the cells that lie across the threshold potential is transferred to cells at the rest potential according to a mapping generated during the pre-processing steps. Details of this mechanism are given by Osborne et al., 2021.

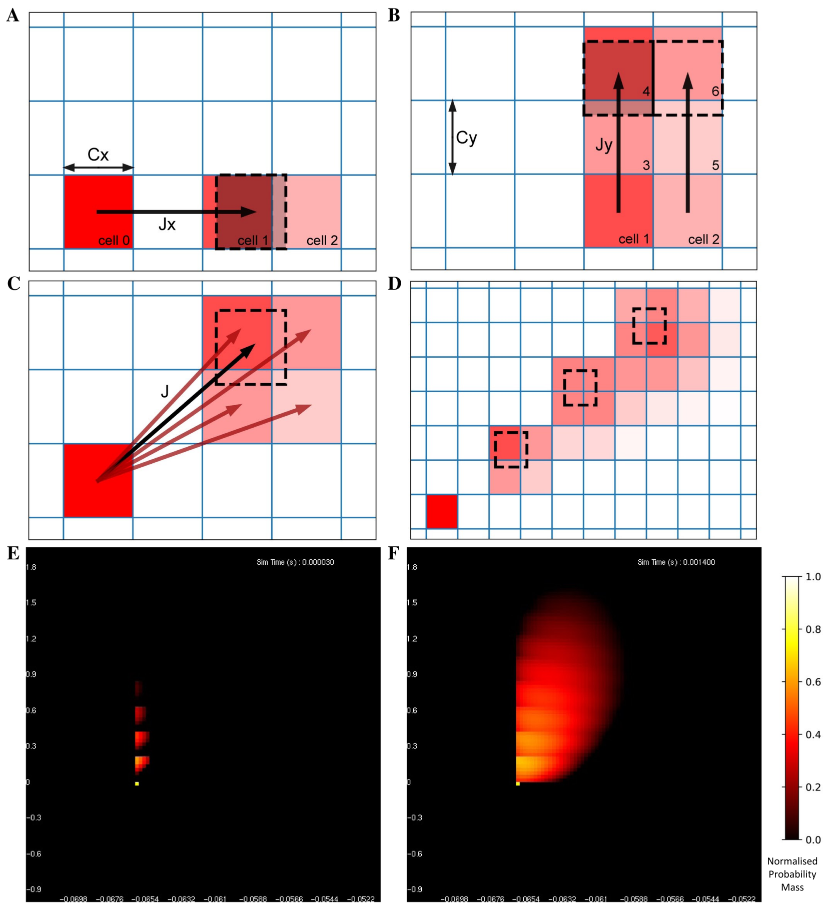


Figure 3.3: (A) The change in state, J, of a neuron due to a single incoming spike can be split into component parts, Jx and Jy for the horizontal and vertical dimensions respectively. All neurons with a state within cell 0 will be translated by Jx due to a single incoming spike. Because all cells are the same width (Cx), the uniformly distributed probability mass of cell 0 will be shared among a maximum of two cells, cell 1 and cell 2. The offset of cell 1 from cell 0 is equal to $floor(Jx/Cx)$ (for negative Jx, it is $ceil(Jx/Cx)$) with cell 2 being the one beyond that. The proportion of mass transferred from cell 0 to cell 1 is equal to $1 - (Cx \,\%\, Jx)$ and the remainder is transferred to cell 2. (B) Once the mass proportions have been calculated in the horizontal direction, the same calculations are made with cells 1 and 2 in the vertical direction using Cy and Jy. The proportion calculated from cell 0 to cell 1 is split between cells 3 and 4. The proportion in cell 2 goes to 5 and 6. (C) The proportions of mass to be transferred from cell 0 to the resulting four cells give an approximation of the effect of transition J. With a constant J, this calculation gives the same relative results for every cell and therefore only needs to be performed once. (D) Iteratively applying the transitions to all cells in the grid spreads mass further across state-space simulating the effect of neurons receiving multiple spikes in a given time step. (E) The probability mass function of a population of leaky integrate-and-fire neurons with an excitatory synaptic conductance rendered in MIIND. The colour of each cell indicates the amount of probability mass. The value has been normalised to the maximum value of all cells. The effect of an incoming spike is to shift mass 0.2 nS/cm^2 in the vertical direction (producing a change in synaptic conductance). At this early point in the simulation, most neurons would have received zero or one spike (indicated by the bright yellow spots) while only a few would have received up to four spikes. (F) As the simulation proceeds, mass continues to be transferred upwards due to incoming spikes but the deterministic dynamics of the model cause mass to also move to the right according to the transitions defined in the matrix file and the population becomes more cohesive.

3.3.2 Extending the grid to N dimensions

An important observation is that the steps shown in Fig. 3.2 for generating the two-dimensional transition matrix file work similarly in higher dimensions. However, the complexity of the algorithm increases significantly. For a three-dimensional underlying neuron model, the grid is extended such that each cell is a cuboid in state space with eight vertices (Fig. 3.4). For an N-dimensional (ND) neuron model, an N-dimensional grid can be constructed with cells made up of 2^N vertices. The task here is to update the calculations involved in the deterministic and non-deterministic processes described above so that they work generically for any number of dimensions. For illustration purposes, we will use a three-dimensional grid.

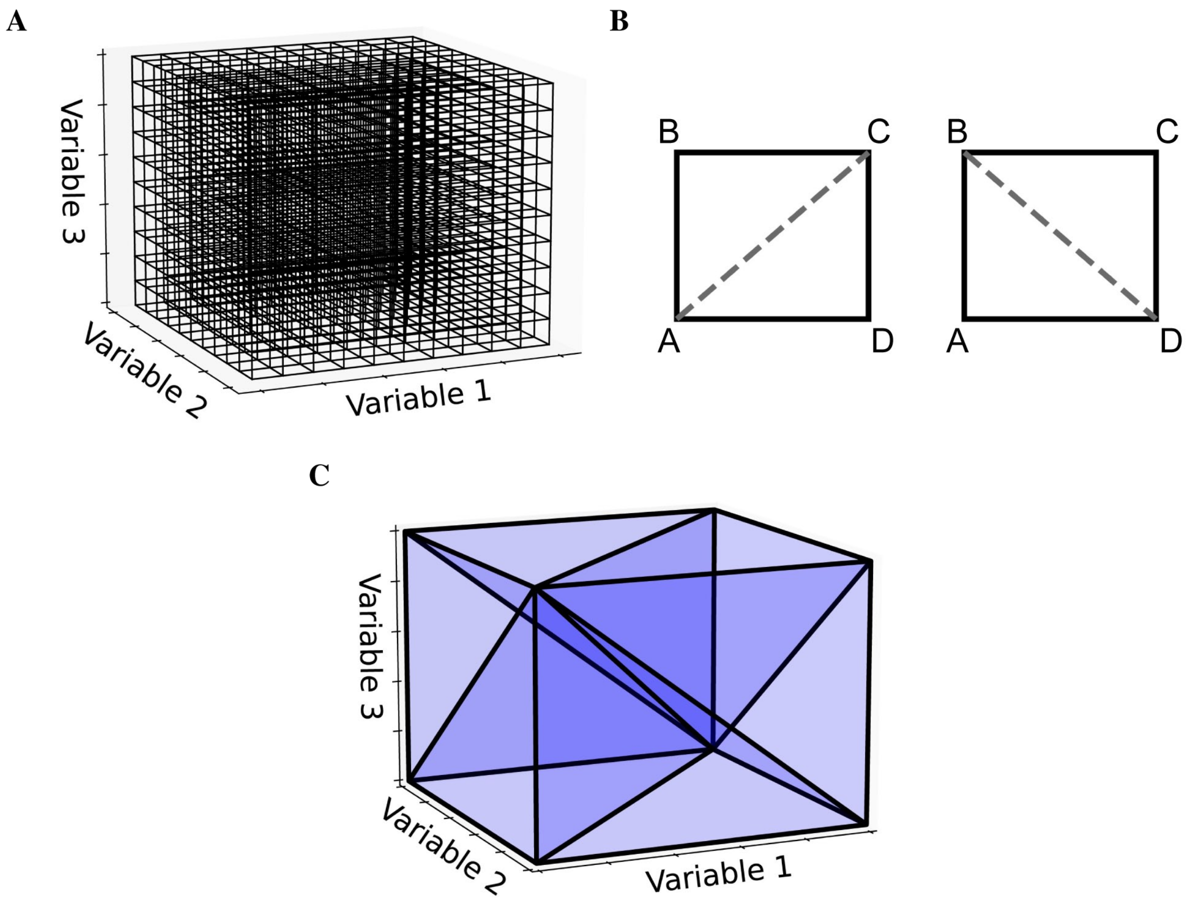


Figure 3.4: (A) With a three-dimensional state space, the grid discretisation is made up of cuboids. (B) For the two-dimensional case, a rectangle has two possible triangulations, [A,B,C] and [A,C,D] or [A,B,D] and [B,D,C]. (C) A cuboid triangulated into six 3-simplices. Other triangulations are possible, some which aim to achieve the minimum number of simplices or to keep the volumes of the simplices as uniform as possible. The Delaunay triangulation makes no guarantees of this kind but is easy to implement and works in N-dimensions.

For the deterministic dynamics, each of the 2^N vertices is again translated according to a single time step of the neuron model and the resulting volume must be triangulated into N-simplices. In three dimensions, a 3-simplex is a tetrahedron. There are many possible triangulations of an N-dimensional cell. As an example, in the simpler two-dimensional case, if the four vertices of a rectangle are labelled A to D in a clockwise fashion as in Fig. 3.4B, the possible triangulations are [A,B,C] and [A,C,D] or [A,B,D] and [B,D,C]. As with the number of possible triangulations, the number of resulting N-simplices increases with dimensionality and there are many algorithms available to generate them (Haiman, 1991). Many algorithms exist to find the so-called Delaunay triangulation of a set of points, which has a specific definition: A set of triangles (or N-simplices) between points such that no point lies within the circumcircle (or hypersphere) of any triangle (or N-simplex) in the set. This definition results in a quite well-formed triangulation (minimising the number of long and thin triangles). One of the simplest ways to find the Delaunay triangulation of a set of points in N dimensions is to use the quickhull algorithm (Brown, 1979; Barber et al., 1996). The initial triangulation of the transformed cell is calculated using this method. To improve efficiency of this triangulation step, instead of finding the Delaunay triangulation for every translated cell, quickhull can be applied once to a unit N-cube as shown in Fig. 3.4C. Under the assumption that the transformed cell remains a convex hull (not unreasonable given that the time step should be small), the triangulation of the unit N-cube can be applied to every transformed cell without re-calculating.

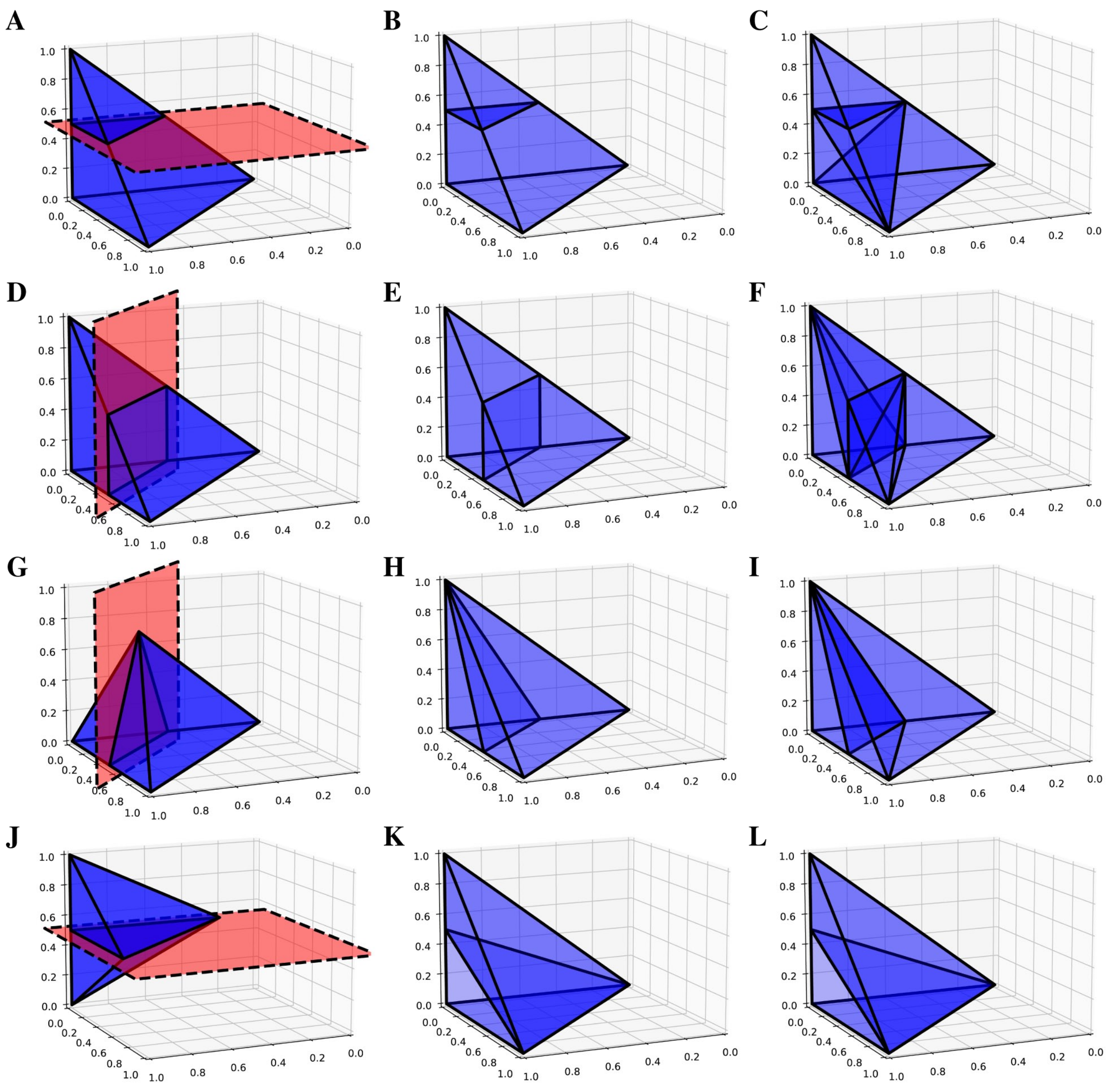


Figure 3.5: (A,D,G,J) Possible plane intersections with a 3-simplex. (B,E,H,K) Illustration of how each intersection is represented in the algorithm such that intersections bisect the relevant edges. (C,F,I,L) The resulting triangulations of the bisected 3-simplex which can be applied to all intersections of this type when calculating the transitions. (A-C) A plane intersection leaving one vertex of the 3-simplex above the plane and three vertices below. (D-F) A plane intersection leaving two vertices on either side of the plane. (G-I) A plane intersection which goes through one of the vertices leaving one vertex above the plane and two vertices below. (J-L) A plane intersection which goes through two vertices leaving one vertex on either side.

As with the two-dimensional version, the next step is to recursively test each N-simplex for intersections with hyperplanes of the grid. Fig. 3.5 shows examples of possible plane intersections of a 3-simplex. Finding an intersection, again, trivially involves checking if vertices lie on both sides of the hyperplane. The new vertices resulting from the intersections with the edges of the N-simplex describe two new shapes on either side of the plane. These must again be triangulated into smaller N-simplices. As with the first triangulation of the unit N-cube, pre-calculated triangulations of a unit N-simplex can be mapped to each newly generated N-simplex of the transformed cell. However, as Fig. 3.5 shows, there are multiple ways that an N-simplex can be bisected with each requiring a different triangulation of the resulting shapes. Each type of intersection can be described uniquely with the number of vertices above the hyperplane, below the hyperplane and on the hyperplane. Table 3.1 gives the possible bisections of a 3-simplex which are illustrated in Fig. 3.5. The terms "above" and "below" are just used here to describe each side of the hyperplane and do not represent a position relative to each other or the hyperplane. Listing 3.1 gives the programmatic way to find all possible intersections of an N-simplex.

Table 3.1: Possible vertex configurations from bisections of a 3-simplex

Vertices above the plane	Vertices below the plane	Vertices on the plane	Resulting new vertices
1	3	0	3
2	2	0	4
1	2	1	2
1	1	2	1

Listing 3.1: Calculate all possible vertex combinations to uniquely identify each type of intersection of an N-simplex

```
For each possible number of co-planar vertices which is between 0 and 2^N - 2:
    List all possible combinations of the remaining vertices above and below the
    ↪ hyperplane excluding 0
```

For each of the vertex combinations which uniquely identifies a type of intersection, the appropriate triangulation of the resulting shapes can be pre-calculated using the Delaunay triangulation of a unit N-simplex. To do this, the vertices of the N-simplex are assigned to be "above", "below" or "on" according to the vertex combination. At this point, no hyperplane exists to test for intersection points. However, we know that edges that pass between an "above" vertex and a "below" vertex will be intersected so we can choose to bisect that edge to produce a new vertex as shown in Fig. 3.5. This represents a good enough approximation of the eventual N-simplex bisection and the quickhull algorithm can be performed on the resulting two shapes. The full dictionary of vertex combinations to triangulations is stored in a lookup table so that, during the actual subdivision of N-simplices in the grid, all that is required is to find the correct intersection in the table and to apply the triangulation mapping. As before, the algorithm continues recursively until no more triangulations are required and the volumes of all N-simplices are summed to calculate the proportion of probability mass which will be shared among the relevant cells.

Solving the non-deterministic dynamics in N dimensions is precisely the same as for two dimensions. In the same way that the probability mass proportion was recursively shared among two new cells per dimension, the resulting number of cells to which mass is transitioned due to a single incoming spike is at most 2^N. No intersections of triangulations are required for this calculation as only the cell width and the jump value in each dimension is required as shown in Fig 3.3. The MIIND algorithm proceeds in the same way as it did for two dimensions. First applying the matrix of transitions for the deterministic dynamics to the grid, then iteratively applying the jump transition to each cell multiple times to approximate the spread of probability mass due to Poisson distributed input. If the underlying neuron model has a threshold-reset mechanism, probability mass in the cells at threshold (for a three-dimensional grid, this is a two-dimensional set of cells) is transferred to a set of reset cells according to another pre-calculated mapping.

3.3.3 Running an ND Simulation in MIIND

When implementing the ND extension to the grid method in MIIND, care has been taken to minimise any changes to how the user builds and runs a simulation. Listing 3.2 shows a MIIND simulation file for defining two neuron populations in an E-I configuration as examined later in section 3.3.5.

Listing 3.2: The XML-style simulation file for an E-I network in MIIND

```
<Simulation>
<WeightType>CustomConnectionParameters</WeightType>
<Algorithms>
<Algorithm type="GridAlgorithmGroup" name="COND3D" modelfile="cond3d.model"
    ↪ tau_refractive="0.002" transformfile="cond3d.tmat" start_v="-65" start_w
    ↪ ="0.00001" start_u="0.00001">
<TimeStep>1e-03</TimeStep>
</Algorithm>
</Algorithms>
<Nodes>
<Node algorithm="COND3D" name="E" type="EXCITATORY" />
<Node algorithm="COND3D" name="I" type="INHIBITORY" />
</Nodes>
<Connections>
```

```
<IncomingConnection Node="E" num_connections="10" efficacy="0.15" delay="0.0"
    ↪ dimension="1"/>
<IncomingConnection Node="I" num_connections="10" efficacy="0.15" delay="0.0"
    ↪ dimension="1"/>

<Connection In="E" Out="E" num_connections="50" efficacy="1" delay="0.003" dimension
    ↪ ="1"/>
<Connection In="I" Out="E" num_connections="50" efficacy="4" delay="0.003" dimension
    ↪ ="2"/>

<Connection In="E" Out="I" num_connections="50" efficacy="1" delay="0.003" dimension
    ↪ ="1"/>
<Connection In="I" Out="I" num_connections="50" efficacy="4" delay="0.003" dimension
    ↪ ="2"/>

</Connections>
<Reporting>
    <Display node="E" />
   <Average node="E" t_interval="0.001" />
   <Average node="I" t_interval="0.001" />
   <Rate node="E" t_interval="0.001" />
   <Rate node="I" t_interval="0.001" />
</Reporting>
<SimulationRunParameter>
<master_steps>10</master_steps>
<t_end>TE</t_end>
<t_step>1e-03</t_step>
<name_log>cond.log</name_log>
</SimulationRunParameter>
</Simulation>
```

The full details of the syntax for a simulation file is provided by Osborne et al., 2021. Little in this file has changed to accommodate higher dimensional neuron models. In the definition of the *Algorithm*, COND3D, the attributes *start_v*, *start_w*, and *start_u* allow the user to define the starting position (of a Dirac delta peak) for the population in the three-dimensional space. Similarly-named attributes can be added for higher dimensions. The *modelfile* and *transformfile* attributes should point to the required pre-processed files generated from the algorithm described in section 3.3.2.

The *Connection* elements describe the inhibitory and excitatory connections between the two populations (nodes) E and I. As discussed earlier, each population simulated using the numerical PDT is influenced by one or more Poisson noise processes which change the probability mass function to approximate each neuron in the population receiving Poisson distributed spike trains. In MIIND, populations interact via their average output firing rate which becomes the rate parameter of the input Poisson process for the target population. Four such connections are set up here. The *num_connections* attribute indicates how many incoming connections each neuron in the target (*Out*) population receives from the source (*In*) population. This has the effect of multiplying the incoming firing rate parameter.

The *efficacy* attribute gives the instantaneous jump value caused by a single incoming spike. The *dimension* attribute has been newly added and gives the direction in which the jump occurs. In this example, spikes from the excitatory population cause a change of 1 nS/cm^2 change in dimension 1 which corresponds to the w variable. Finally, the *delay* attribute gives the transmission delay of the instantaneous firing rate between populations which allows MIIND to simulate the complex dynamics which can arise when this is a non-zero value.
All other aspects of the file remain unchanged though the *Display* element which tells MIIND to render the probability mass function of population E during the simulation now causes a three-dimensional rendering of the function in state space. For higher dimensions, which three dimensions to display can be chosen during simulation.
The main change to MIIND to support ND neuron models is the addition of the *generateNdGrid* method in the MIIND Python module. Listing 3.3 shows a function set up in Python, *cond*, which describes the time evolution of a LIF neuron with excitatory and inhibitory conductances. The *generateNdGrid* method generates the *cond3d.model* and *cond3d.tmat* support files which are referenced in the simulation file above (listing 3.2). The method takes as parameters:

1. The Python function defining the model dynamics.
2. The name of the generated files.
3. The minimum values in state space.
4. The span of the grid in state space.
5. The resolution of the grid.
6. The threshold potential.
7. The reset potential.
8. Any additional change in state of a neuron after being reset to the reset potential (in this case, there is none).
9. The timescale of the neuron model in seconds.
10. The time step with which to solve the neuron model in seconds.

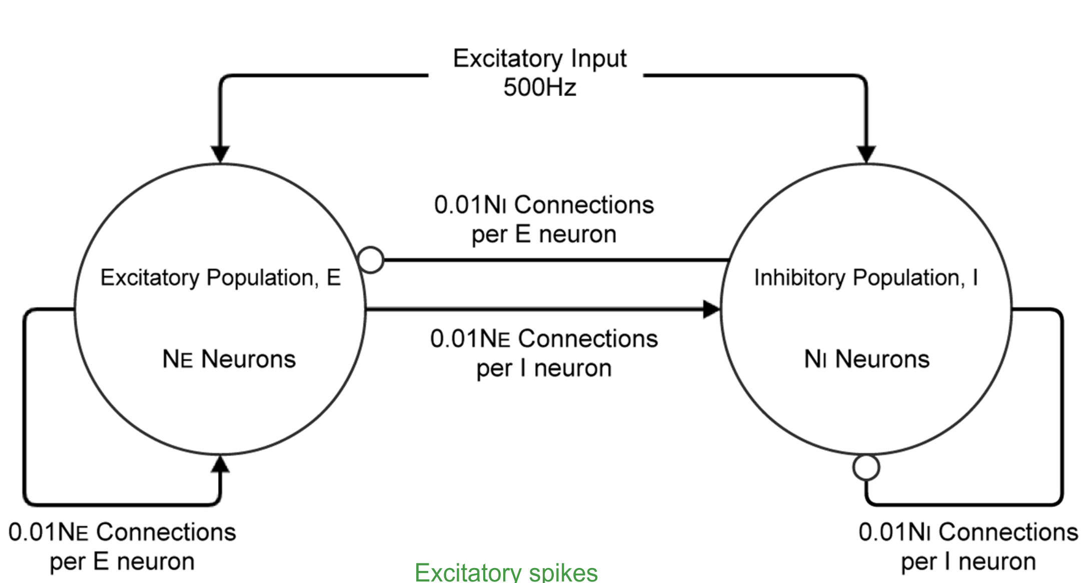
Listing 3.3: An example Python script to generate the support files for a three-dimensional LIF neuron population in MIIND.

```python
import miind.miindgen as miindgen

def cond(y):
    V_l = -70.6
    V_e = 0.0
    V_i = -75
    C = 281
    g_l = 0.03
    tau_e = 2.728
    tau_i = 10.49

    v = y[2]
    w = y[1]
    u = y[0]

    v_prime = (-g_l*(v - V_l) - w * (v - V_e) - u * (v - V_i)) / C
    w_prime = -(w) / tau_e
    u_prime = -(u) / tau_i

    return [u_prime, w_prime, v_prime]

miindgen.generateNdGrid(cond, 'cond3d', [-0.2,-0.2,-80], [5.4,5.4,40.0], [50,50,50],
    ↪ -50.4, -70.6, [0.0,0.0,0.0], 1, 0.001)
```

Running a script such as this performs the steps outlined in section 3.3.2. To see further examples of ND simulations in MIIND, once the software has been installed (using **pip install miind**), the *examples/model_archive* directory of the MIIND repository contains the required files for a number of different three- and four-dimensional neuron model populations. The three experiments presented below are available in the *examples/miind_nd_examples* directory of the MIIND repository.

3.3.4 Testing a Single Population

Initially, a single population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductance variables was simulated in MIIND and compared to a so-called Monte Carlo approach. The definition of the underlying neuron model is given in equation 3.1 and the parameters are listed in table 3.2. v represents the membrane potential, u represents the conductance of inhibitory synapses which will increase with increased inhibitory input. w represents the conductance of the excitatory synapses. C is the membrane capacitance and g_l is the leak conductance. V_l, V_e, and V_i are the reversal potentials for their respective conductances. The refractory period, during which the state is held constant at the reset potential, has been set to 2 ms. Fig. 3.6 shows a schematic of the neuron model state space in three dimensions and the effect of excitatory and inhibitory input spikes. Due to the dynamics of the model, mass in cells with a high u value will move to lower values of v and mass at high w values will move to higher cells in v.



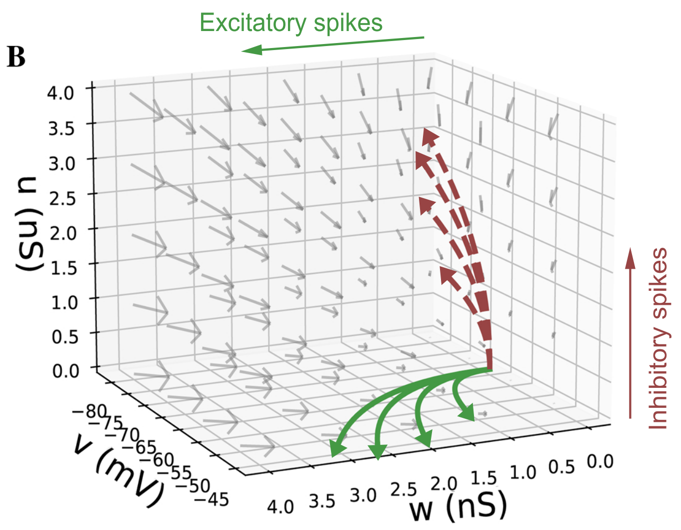


Figure 3.6: (A) A schematic of the E-I population network. The excitatory population, E is made up of N_E neurons. The inhibitory population, I contains $N_I = 10000 - N_E$ neurons. Each population receives an excitatory external input of 500Hz. Each neuron in both populations receives $0.01N_E$ excitatory connections and $0.01N_I$ inhibitory connections. Arrows represent an excitatory connection, circles represent an inhibitory connection. (B) The three-dimensional state space of the leaky integrate-and-fire neuron with an excitatory and inhibitory synaptic conductance. v is the membrane potential, w is the excitatory synaptic conductance, and u is the inhibitory synaptic conductance. The vector field shows the direction of motion in state-space for neurons with no external impulse. Neurons which receive an excitatory input spike are shifted higher in w. Neurons which receive an inhibitory input spike are shifted higher in u. The solid curves show trajectories of neurons under excitatory impulse alone. The dashed curves show trajectories of neurons under inhibitory impulse alone.

Table 3.2: Parameters used for equations 3.1 and 3.2

Parameter Name	Values and Notes
Equation 3.1	Leaky integrate-and-fire neuron with an excitatory and inhibitory synaptic conductance
g_l	0.03 nS/cm^2
E_l	-70.6 mV
E_e	0.0 mV
E_i	-75 mV
C	281 pF/cm^2
τ_e	2.728 ms
τ_i	10.49 ms
Refractive period	2 ms
Threshold potential	-50.4 mV
Reset potential	-70.6 mV
Equation 3.2	Hodgkin-Huxley Neuron
g_l	0.5 mS/cm^2
g_k	30 mS/cm^2
g_{na}	100 mS/cm^2
V_k	-90 mV
V_{na}	50 mV
V_l	-65 mV
C	1.0 μF/cm^2
α_m	$0.32(13 - v + V_t)/(e^{\frac{13-v+V_t}{4}} - 1)$
α_n	$0.032(15 - v + V_t)/(e^{\frac{15-v+V_t}{5}} - 1)$
α_h	$0.128e^{\frac{17-v+V_t}{18}}$
β_m	$0.28(v - V_t - 40)/(e^{\frac{v-V_t-40}{5}} - 1)$
β_n	$0.5e^{\frac{10-v+V_t}{40}}$
β_h	$4/(1 + e^{\frac{40-v+V_t}{5}})$
V_t	-63 mV

$$C\frac{dv}{dt} = -g_l(v - V_l) - w(v - V_e) - u(v - V_i)$$
$$\tau_e\frac{w}{dt} = -w$$
$$\tau_i\frac{u}{dt} = -u$$

$$v > threshold \longrightarrow v = reset \tag{3.1}$$

The Monte Carlo simulation was set up in Python for a population of 10000 neurons following the dynamical system in equation 3.1. For a time step of 1ms, neurons receive a number of input spikes sampled from a Poisson distribution with a given rate parameter. Each spike causes a 1.5 nS/cm^2 increase in the excitatory synaptic conductance variable, w. Each neuron also receives excitatory and inhibitory Poisson noise at 50Hz, again, with each excitatory spike causing a 1.5 nS/cm^2 increase in w and each inhibitory spike causing a 1.5 nS/cm^2 increase in u. Both u and w were set to 0 nS/cm^2 at the start of the simulation.

A MIIND simulation was similarly set up. Six separate grid transition files were generated all according to equation 3.1 but with different grid resolutions: 50x50x50 (for u, w, and v respectively), 100x100x100, 150x150x150, 100x100x200, 200x200x100, and 50x50x300. For all resolutions, the grid spans the model state space for u = -0.2 nS/cm^2 to 5.2 nS/cm^2, w = -0.2 nS/cm^2 to 5.2 nS/cm^2, and v = -80 mV to -40 mV. These ranges represent the limits of the values that the variables can take in the MIIND simulation but were chosen because all significant probability mass is contained in this volume throughout. All simulations produced 1.2 seconds of activity. The average membrane potential, synaptic conductances, and firing rate of the population were recorded.

Though MIIND has not been fully benchmarked, it is instructive to see the relative benefits to computational efficiency with differing grid resolutions. For the grid resolutions, 50x50x50, 100x100x100, 150x150x150, and 50x50x300, the time from starting the MIIND program to the beginning of the simulation was recorded to give an indication of the effect of load times with greater transition file sizes. Then the time to complete the simulation was recorded. The same simulation from above was performed without recording the membrane potential or firing rate to the hard drive. The machine used to produce the results has a solid-state drive (SSD), an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, and an NVidia Geforce GTX 1060.

3.3.5 An E-I Network

To demonstrate how MIIND is able to simulate the interaction of multiple populations and capture changes in behaviour with different parameters, a population network was set up in an E-I configuration

(Brunel, 2000). Fig.3.6 shows the population level connections. In both the MIIND and Monte Carlo simulations, for each connection, the average firing rate of the source population is used as the rate parameter for the Poisson input to the target population. The Monte Carlo simulation was set up in Python for 10000 neurons following the dynamics of equation 3.1. Parameters for the neuron model and E-I network model are adapted from Sukenik et al., 2021. The 10000 neurons are shared among the two populations according to a ratio parameter of excitatory to inhibitory neurons. That is, the number of inhibitory neurons, N_I was chosen and the number of excitatory neurons, N_E was set equal to $10000 - N_I$. The excitatory and inhibitory conductance jump values are held constant and a weight is multiplied by the Poisson rate parameter of each connection to reflect that each neuron should receive $0.01N_E$ excitatory connections and $0.01N_I$ inhibitory connections. A transmission delay of 3 ms is applied to all inter-population connections. Finally, each population receives a 500Hz excitatory Poisson distributed input with each spike causing a 1.5 nS/cm^2 jump in w. Table 3.3 gives the full list of parameters for the E-I model. MIIND was set up, in the same way, using a newly generated grid with resolution 150x150x150. The grid for this simulation covers a much larger volume of state space as it is expected that there will be large fluctuations in the conductance variables. Therefore, the size of the grid was set to u = -10 nS/cm^2 to 100 nS/cm^2, w = -5 nS/cm^2 to 25 nS/cm^2, and v = -80 mV to -40 mV. Across simulation trials, all parameters were kept constant except for N_I.

3.3.6 A Four-dimensional Neuron Population

To test the performance of MIIND with populations of four-dimensional neurons, we simulated a population of Hodgkin-Huxley neurons (Hodgkin and Huxley, 1952). This gold-standard model has not been simulated with a population density approach before. A fourth time-dependent variable significantly increases the amount of computation required to generate the transition matrix and its size beyond the three-dimensional case above. As before, a Monte Carlo simulation was set up for comparison. The Hodgkin-Huxley neuron model is defined in equation 3.2. As in equation 3.1, the neuron has a capacitance, C, and a leak conductance, g_l, with reversal potential, V_l. The potassium and sodium synaptic conductances, g_k and g_{na} remain constant with respective reversal potentials, V_k and V_{na}. However, they are modulated by the three time-dependent gating variables, n, m, and h. The definitions of α and β are given in table 3.2.

Table 3.3: Parameters used for the E-I network model

Parameter Name	Values and Notes
	Parameters apply to both the MIIND and Monte Carlo simulations
External firing rate	500 Hz to both E and I populations
External excitatory jump	1.5 nS/cm^2 change in w per incoming spike
N_I	Free parameter in the range 1000 - 9000
N_E	$10000 - N_I$
Number of E to E connections	$0.01N_E$
Number of E to I connections	$0.01N_E$
Number of I to I connections	$0.01N_I$
Number of I to E connections	$0.01N_I$
Excitatory jump for E to E connections	1 nS/cm^2 increase in w per incoming spike
Excitatory jump for E to I connections	1 nS/cm^2 increase in w per incoming spike
Inhibitory jump for I to I connections	4 nS/cm^2 increase in u per incoming spike
Inhibitory jump for I to E connections	4 nS/cm^2 increase in u per incoming spike
E to E transmission delay	3 ms
E to I transmission delay	3 ms
I to I transmission delay	3 ms
I to E transmission delay	3 ms

$$\begin{aligned}
C\frac{dv}{dt} &= -g_k n^4(v - V_k) - g_{na} m^3 h(v - V_{na}) - g_l(v - V_l) \\
\frac{m}{dt} &= \alpha_m(1 - m) - \beta_m m \\
\frac{n}{dt} &= \alpha_n(1 - n) - \beta_n n \\
\frac{h}{dt} &= \alpha_h(1 - h) - \beta_h h.
\end{aligned} \tag{3.2}$$

The population was given a Poisson distributed input at various rates between 0 and 40 Hz. The number of input connections to each neuron in the population was set at 100 and can be considered a weight so that the incoming rate would be multiplied by this amount. Each incoming spike produces a 3 mV jump in membrane potential. For MIIND, only one Hodgkin-Huxley grid was generated with dimensions 50x50x50x50 for h, n, m, and v respectively. This resolution was chosen to keep the total number of cells low. The size of the grid was set between -0.1 and 1.1 for the gating variables, and v = -100 mV to 60 mV.

3.4 Results

3.4.1 A Single Population of Three-dimensional Neurons

Fig. 3.7 shows the probability mass functions for six different simulations of a population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances. Each cell has a colour/brightness and an alpha or transparency value such that cells with a higher probability mass are a brighter yellow, and more opaque than cells with lower probability mass which are darker red and more transparent. This plotting style allows the centre of the function volume to be seen from the outside. Cells with zero probability are entirely transparent so that only significant cells are visible. Due to the greater opacity which often appears in the central volume of the function, the MIIND user may also rotate the entire volume to view the function from all angles. In Fig. 3.7A and B, when only an excitatory input is provided, the function remains in the two-dimensional plane at $u = 0$ and is the same function as produced in the purely two-dimensional model demonstrated in de Kamps et al., 2019 and Osborne et al., 2021. Likewise, when only an inhibitory input is provided (Fig. 3.7C and D), the function stays at $w = 0$. Fig. 3.7E and F show the result of both an excitatory and inhibitory input. When enough excitatory input is provided, probability mass reaches the threshold membrane potential and is reset causing a sharp cut-off at those values. The brighter yellow cells in the centre of the function's volume indicate that the majority of neurons can be found there travelling from the reset to threshold potential receiving close to the average number of excitatory and inhibitory input spikes. Further out, at higher values of u and w, the probability of finding a neuron reduces as neurons are less likely to receive many more spikes than average.

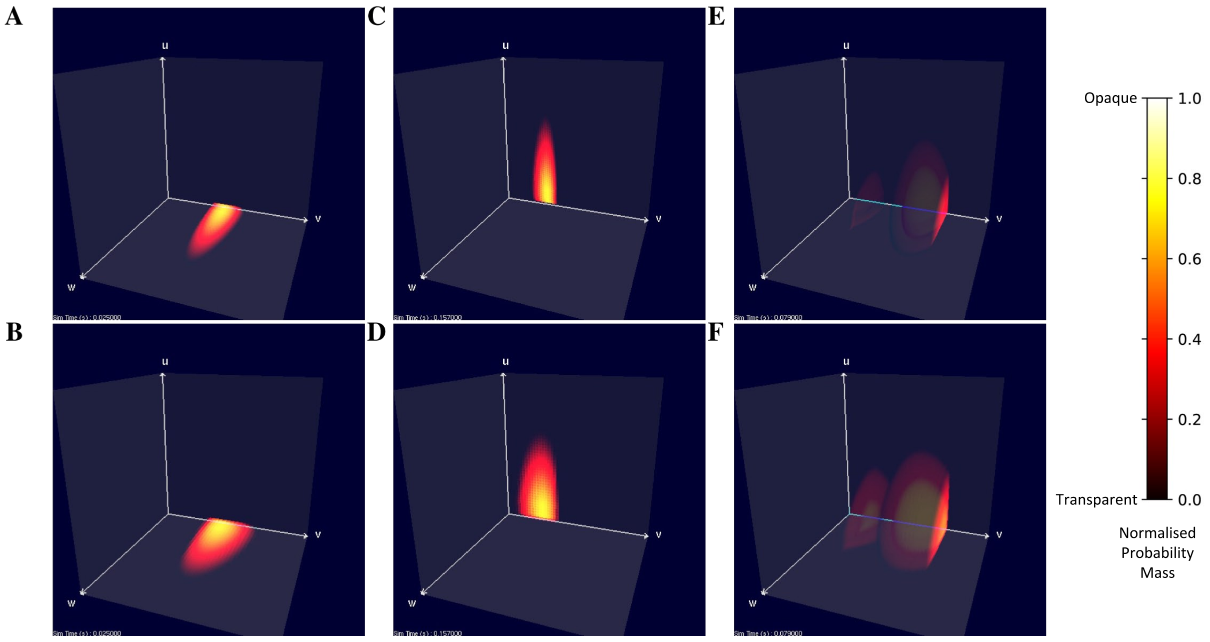


Figure 3.7: Visualisations of a population of leaky integrate-and-fire neurons with an excitatory and inhibitory synaptic conductance in MIIND. Cells with no probability mass are transparent. With increasing probability mass, they become more opaque and change from red to yellow. The colour and opacity are normalised to the value of the cell with the highest probability mass. (A,C,E) The probability mass function across a 150x150x150 grid. (B,D,F) The probability mass function across a 50x50x50 grid for the same simulation time as the image above. (A-B) When the population receives only excitatory incoming spikes, the probability mass function remains in the plane at $u = 0$. (C-D) When the population receives only inhibitory incoming spikes, the probability mass function stays in the plane at $w = 0$. (E-F) When the population receives both inhibitory and excitatory incoming spikes, the probability mass function extends into the state space. In this case, the excitatory input is enough to overcome the inhibitory input and the mass function moves across the threshold potential. The bright face shows the probability mass at the threshold. Probability mass which has been reset reappears at the reset potential and moves further into the state space.

Fig. 3.8 shows average membrane potential recorded from multiple simulations of a population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances. The scatter points show the average potential of 10000 individual neurons simulated using the Monte Carlo approach. The remaining curves show the average potential of populations simulated in MIIND using 3-dimensional grids of different resolutions. For the transient period before the membrane potential reaches a steady-state, all the MIIND simulations remain synchronised with the Monte Carlo results. As would be expected, the least accurate result comes from the lowest resolution grid, 50x50x50. However, even at this resolution, the mean error between the Monte Carlo activity and the MIIND result is only 0.354 mV. The error is reduced significantly for 100x100x100 (0.115 mV) then further reduced but only slightly for 150x150x150 (0.063 mV) suggesting a degree of diminishing return for increasing the resolution in an equal fashion across dimensions. The error from the 200x200x100 grid is the same as the 100x100x100 grid but the 100x100x200 grid does better (0.059 mV) indicating that increasing the resolution of the membrane potential dimension is a more efficient way to attain accurate results for this underlying neuron model. To illustrate this further, the 50x50x300 grid performs the best of the trials with an average error of 0.054 mV despite the low resolution of the conductance dimensions. Over a range of average rates (Fig. 3.8B) of the Poisson distributed input, the steady-state membrane potential of the MIIND simulations, again, approaches those of the Monte Carlo results with increasing resolution. For low input rates, when the majority of neurons are subthreshold, the 150x150x150 grid gives the closest approximation to the Monte Carlo results. However, once the majority of neurons are crossing the threshold and firing, the 50x50x300 grid gives better agreement. Fig. 3.8C shows the average excitatory conductance variable for the grids across the range of lower input rates (1-10Hz) in comparison to the Monte Carlo approach. The 50x50x300 grid underestimates the conductance which could account for the underestimation of the membrane potential for the same input. The 150x150x150 grid, by contrast, has better agreement with the membrane potential and excitatory conductance for these rates which produce mostly sub-threshold activity in the population.

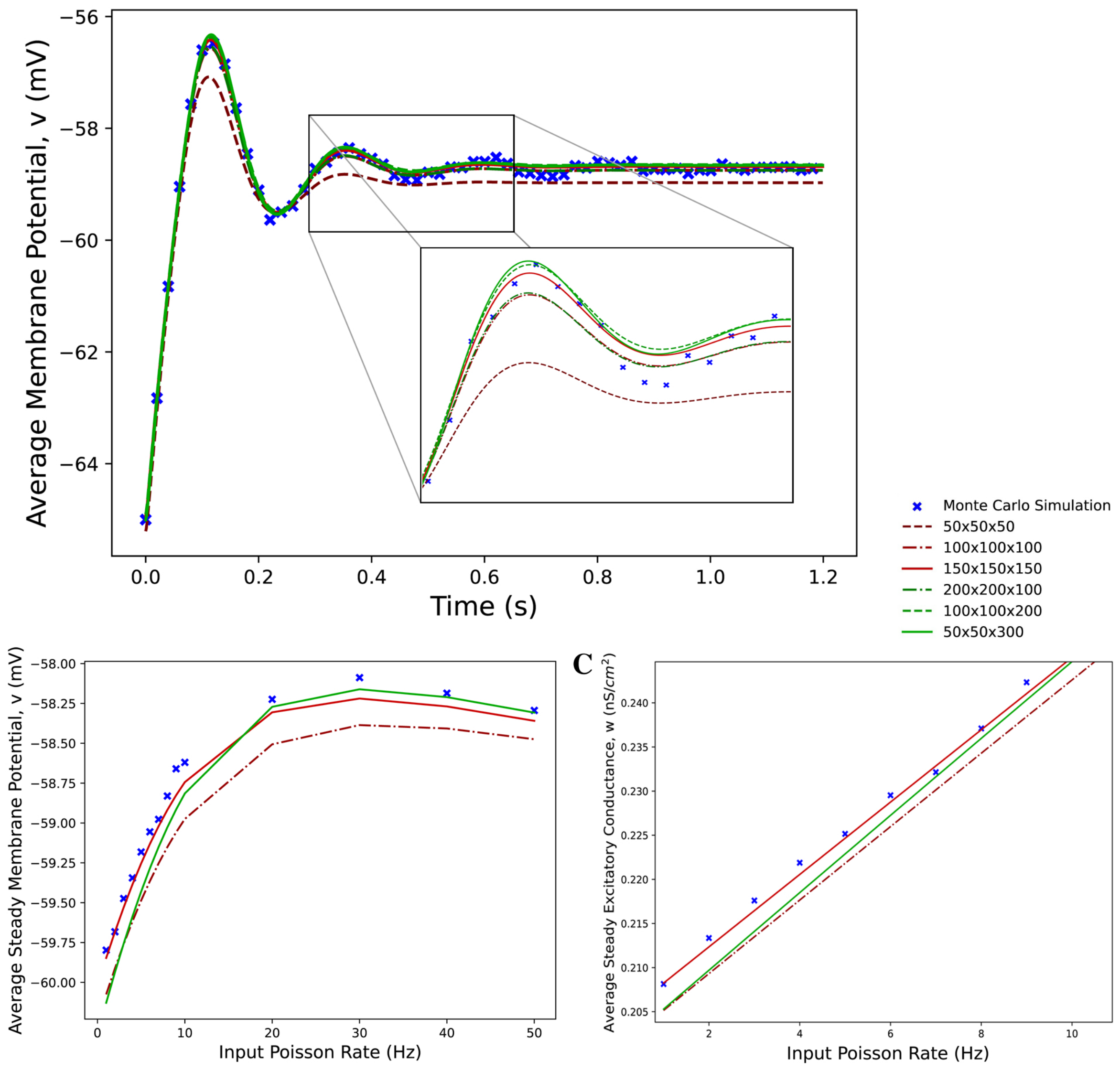


Figure 3.8: (A) The average membrane potential for a single population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances simulated using a Monte Carlo approach and using MIIND with grids of different resolutions. (B) The effect on the average steady-state membrane potential with different rates of the Poisson distributed input for the Monte Carlo simulation and different MIIND grid resolutions. (C) The effect on the average steady-state excitatory conductance variable with increasing Poisson input rate. Only the mean of the values for the Monte Carlo simulation are shown here (without a variance or standard deviation) because the MIIND simulation produces no such statistic and so no comparison can be made.

Fig. 3.9 shows the average firing rates of the same Monte Carlo and MIIND simulations. The differences in grid resolution produce a similar trend in error, with the lowest resolution, 50x50x50 laying furthest away from the Monte Carlo simulation and the 50x50x300 grid the closest. However, even at lower resolutions, all the average firing rates of the MIIND populations are very well matched to the direct simulation.

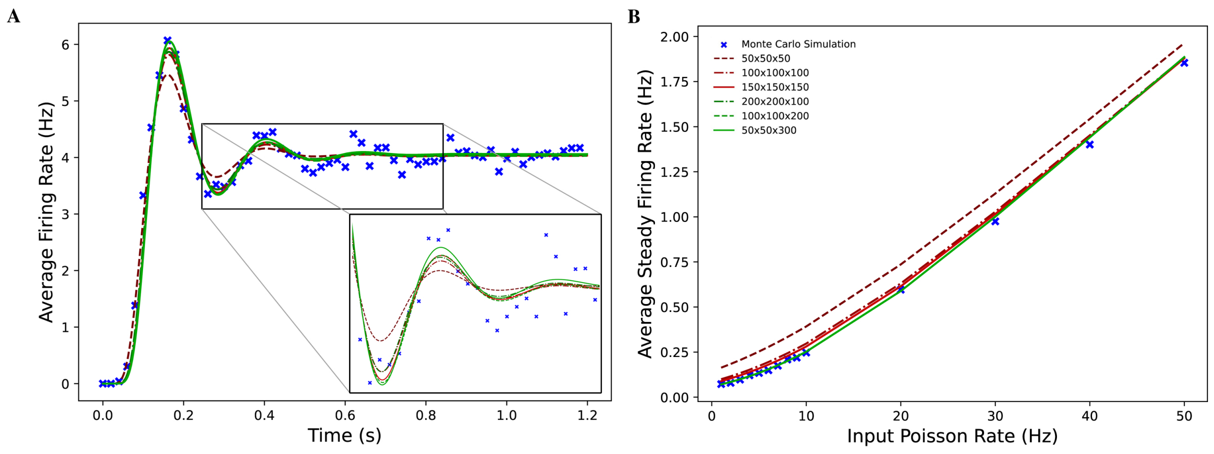


Figure 3.9: (A) The average firing rate of a single population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances simulated using a Monte Carlo approach and using MIIND with grids of different resolutions. (B) The effect on the average steady state firing rate of the population with increasing rate of the Poisson distributed input.

3.4.2 Simulation Speed for Different Grid Resolutions

Table 3.4 shows the load and simulation times for one second of a single population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances. As expected, as the total number of cells increases the load times and simulation times increase. When running multiple short simulations, the load time becomes a significant consideration. However, only the simulation time is dependent on the required length of the simulation. The load time remains constant.

Table 3.4: Times to simulate one second of a population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances in MIIND using different grid resolutions

Grid resolution	Time to load the grid (s)	Time to run the simulation (s)
50x50x50	4.82	2.71
100x100x100	35.58	15.18
150x150x150	126.01	48.7
50x50x300	27.62	11.93

3.4.3 Three-dimensional Neurons in an E-I Population Network

For the Monte Carlo simulation of 5000 excitatory and 5000 inhibitory neurons (with an average of 50 excitatory and 50 inhibitory incoming connections to each), the two populations reach an equilibrium state after an initial transitory phase. Fig 3.10A shows excellent agreement between the average membrane potentials from the two approaches. The initial oscillation in the transient period covers nearly 100 nS/cm^2 in u and 12 nS/cm^2 in w which requires a much larger volume of state space than the single population simulation because of the large synaptic efficacies and recurrent connections involved in the E-I network. In MIIND, as the oscillations reduce, the state space covered by the probability mass function reduces and is therefore discretised by fewer cells. In other words, the cell density covering the function is lower. However, this only causes a minimal amount of additional damping to the oscillation as the function reaches equilibrium.

In the Monte Carlo simulation, with 8000 excitatory neurons and 2000 inhibitory neurons, both populations produce an oscillating pattern as shown in Fig. 3.10B. In the MIIND simulation, in order to match the connection ratios between populations, the number of excitatory and inhibitory connections is set to 80 and 20 respectively. The simulation is also able to produce a similar oscillatory pattern. As would be expected, the population density approach produces a regular oscillation while the Monte Carlo has some variation in the length and amplitude of each oscillation. The double peak of each oscillation can be explained by observing the probability mass function in MIIND during the simulation (Fig. 3.10C-H). The initial peak is produced as the whole population depolarises and approaches the threshold potential. As mass begins to pass the threshold, the reset mass brings the average membrane potential back down. The probability mass is pushed higher in w and u as the recurrent excitatory input and inhibitory input from the other population increase. The excitatory input has the strongest effect on the probability mass close to the reset potential which begins to push the average membrane potential back up towards a second peak. The inhibitory input has the strongest effect on the probability mass close to threshold and less and less mass reaches threshold. The split probability mass function coalesces once more and the cycle can repeat.

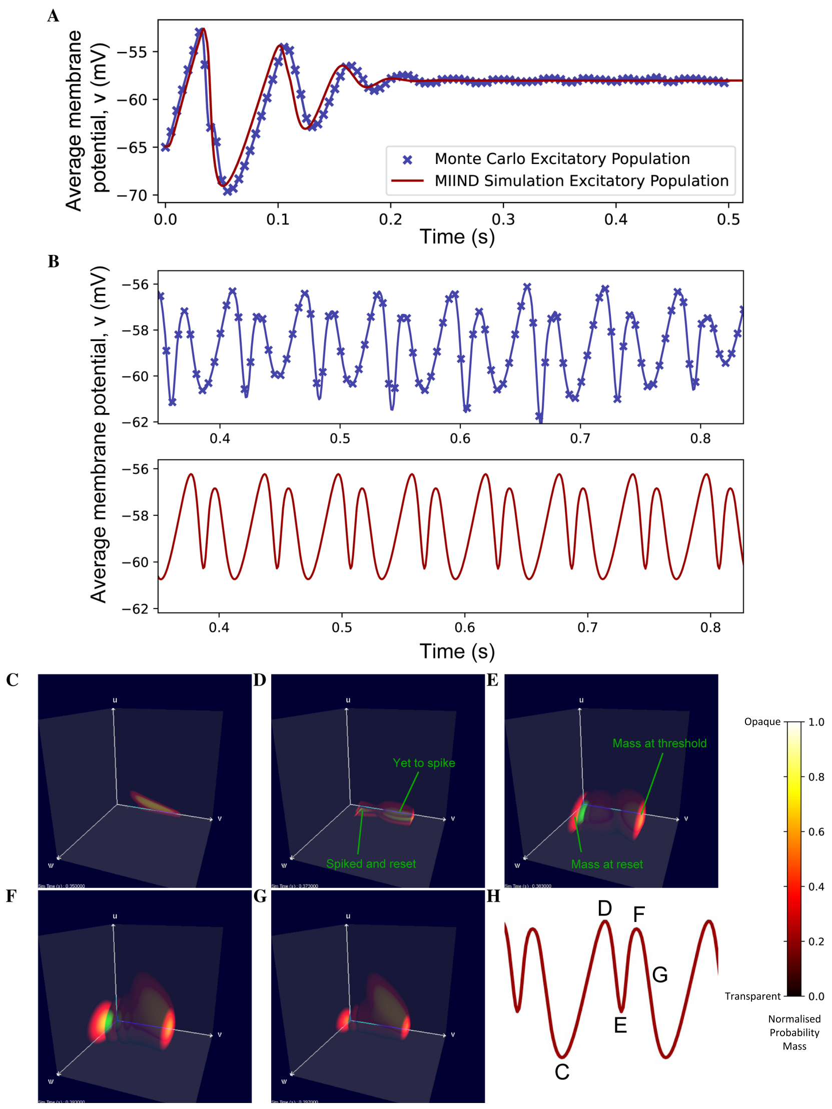


Figure 3.10: (A) The average membrane potential of the excitatory population in the E-I network with a ratio of 1:1 excitatory and inhibitory neurons ($N_E = N_I$). In the MIIND simulation, each connection between populations has the "number of connections" value set to 50. (B) The average membrane potential of the excitatory population in the E-I network with a ratio of 8:2 excitatory to inhibitory neurons ($N_E = 8000, N_I = 2000$). In the MIIND simulation, the excitatory connections have the "number of connections" value set to 80 and the inhibitory connections have the "number of connections" value set to 20. For clarity, the traces from the Monte Carlo simulation and the MIIND simulation have been separated. (C-H) The probability mass function for the excitatory population in MIIND during the double-peaked oscillation with a connection ratio of 8:2. (H) shows the corresponding points in the oscillation. At (C), The population only experiences the external input of 500Hz and is pushed towards the threshold. At (D), though some probability mass has passed threshold and been reset, the majority is close to the threshold and so the average membrane potential is at a peak. At (E), probability mass has continued to cross the threshold so that now a large amount is near the reset potential which brings the average back down. The function has also shifted higher in w due to the excitatory self-connections and more probability mass is pushed across threshold. The function also begins moving upwards in u from the increased inhibitory input but this is not enough to overcome the excitation. At (F), the inhibitory input has continued to push the probability mass function higher in u and much less probability mass now crosses the threshold. At (G), the function continues to shift back away from threshold approaching (C) once again.

When the ratio of excitatory to inhibitory neurons is 9:1, the Monte Carlo simulation demonstrates how the excitatory self-connection causes the excitatory population activity to "blow-up" such that the excitatory conductance reaches a maximum value and neurons fire at their maximum rate. This state is a challenge for MIIND to emulate. Firstly, the number of iterations required to solve the Poisson master equation each time step must be increased to 1000 due to the instability caused by such high firing rates. Secondly, the excitatory conductance variable frequently approaches 80 nS/cm^2 and so the grid must cover a large amount of state-space requiring an unreasonable resolution in the w dimension to maintain the same cell density as previous simulations.

3.4.4 A Single Population of Four-dimensional Hodgkin-Huxley Neurons

Even with a low resolution of 50x50x50x50, MIIND is able to simulate the probability mass function of a population of Hodgkin-Huxley neurons and achieve good agreement with the transient activity and steady-state membrane potential of an equivalent Monte Carlo simulation (Fig. 3.11A). Fig. 3.11B shows how, for a range of input firing rates, the resulting average membrane potential at steady-state approximates that of the Monte Carlo simulations better for higher frequencies. At low input rates, the membrane potential is overestimated. MIIND displays the probability mass function for three of the four dimensions at a time as shown in Fig. 3.11C. With a key press, the user can change the order of dimensions displayed and see any combination of variables. Fig. 3.11F shows the three gating variables, m, n, and h.

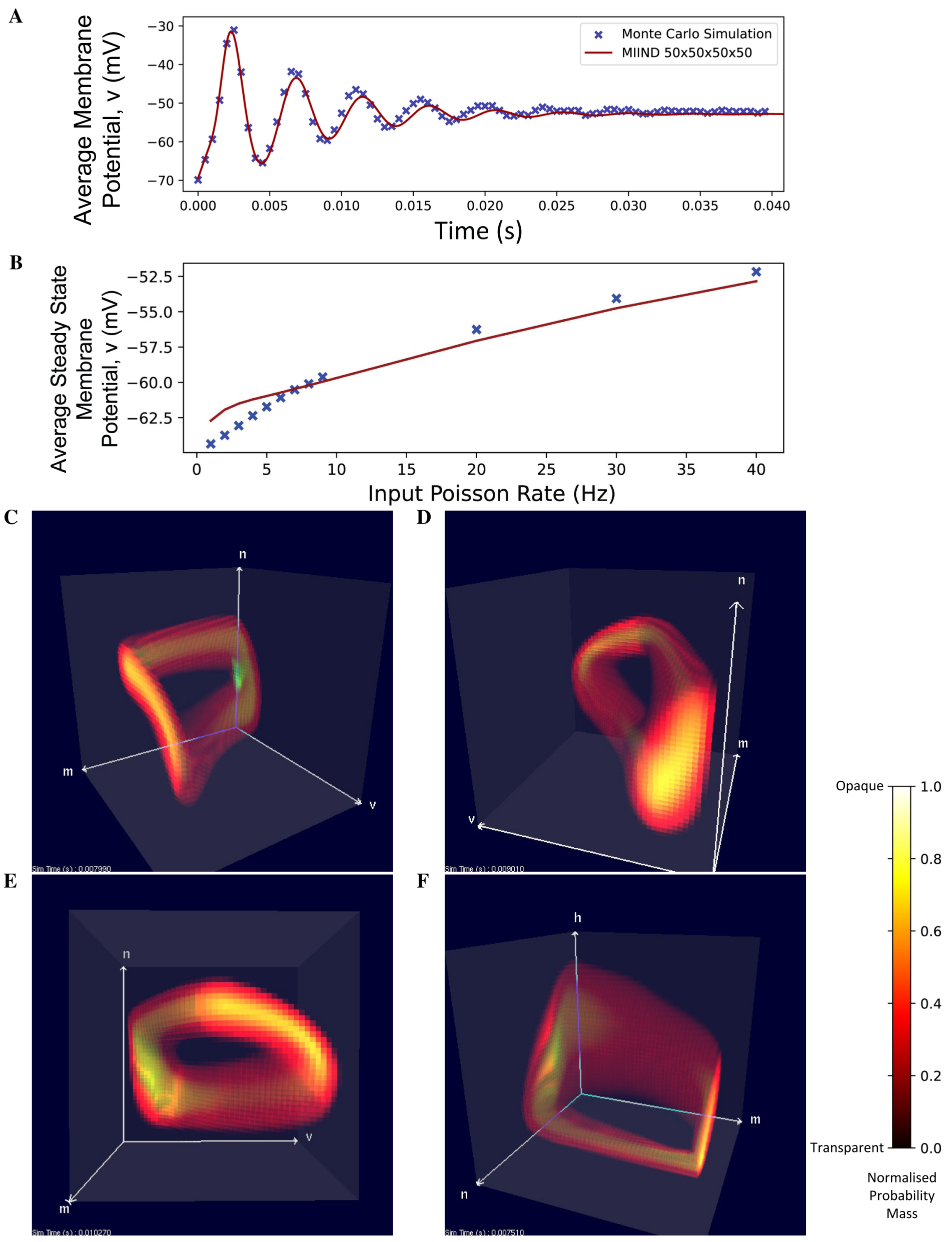


Figure 3.11: (A) The average membrane potential of a single population of Hodgkin-Huxley neurons simulated using a Monte Carlo approach and in MIIND with a 50x50x50x50 grid. (B) The average steady state membrane potential with different rates of the Poisson distributed input. (C-E) The three-dimensional marginal probability mass function of the four-dimensional Hodgkin-Huxley neuron population in MIIND having reached a steady state. The membrane potential v, sodium activation variable m, and potassium activation variable n are shown. (F) The three-dimensional marginal probability mass function showing the gating variables, w, n and h (sodium inactivation variable) only.

3.5 Discussion

The original motivation for applying a population density approach to simulate neurons was to reduce the computational complexity when analysing a large population of homogeneous neurons. This has since been made somewhat redundant with the development of more powerful computers and especially the use of GPGPU architectures. For example, GeNN (Yavuz et al., 2016; J. C. Knight et al., 2021) can simulate in real-time the well-known Potjans-Diesmann microcircuit model (Potjans and Diesmann, 2014) which comprises around 10000 neurons. The analytical solution for the behaviour of a leaky integrate-and-fire population developed by Omurtag et al., 2000 using the diffusion approximation would undoubtedly prove efficient, requiring only a single calculation per population per time interval. However, it would require a lot of manual work to define the full population network and if a different neuron model were to substitute the integrate-and-fire neuron, the entire solution would need to be re-derived. Even with newer techniques such as the refractory density approach, work is required to get the underlying neuron model in a form that can be processed. MIIND uniquely overcomes this limitation allowing the user to define the neuron model without any further manual process to produce a numerical solution to the population density approach. However, solving the master equation for the non-deterministic noise component of the dynamics requires repeated applications of the jump transitions shown in Fig. 3.3. As discussed by Osborne et al., 2021, depending on the model, the time step, and the input firing rate, solving the master equation can require tens or hundreds of iterations per time step of the simulation. This was the case for the EI network in the 8:2 ratio. The sharp changes in firing rate of the two populations combined with large synaptic conductance jumps meant that solving the master equation required 100 iterations per cell per time step to remain stable. The numerical population density approach in MIIND should, therefore, not be used for simulations where computational speed is the most important factor. However, it has been shown (de Kamps et al., 2019) that there is at least an order of magnitude improvement in memory consumption over direct simulation techniques, such as that of NEST, as there is no requirement to store the spike history.

Although computational efficiency is not the primary reason for using the population density approach, there are some benefits to generating the probability mass function over a direct simulation of individual neurons. The probability mass function can be considered the idealised distribution of neuron states. Cells in the grid which have zero mass correspond to volumes of state space where neuron states should never appear. This can be difficult to approximate with a direct simulation of individual neurons for parts of the distribution with a low but non-zero probability mass. Inconsistencies between the behaviour of real neurons and a model could be identified more effectively by comparing to the probability mass function. Also due to the idealised probability mass function, the output metrics of a population such as average firing rate and average membrane potential have no variation due to noise or a specific realisation of the Poisson distributed input. Therefore, no averaging or smoothing is required to produce more readable results as would be expected from a direct simulation (Fig. 3.8 and Fig. 3.9). In the E-I network, 10000 Monte Carlo neurons were enough to produce a similar result to the MIIND simulation. But when that number is reduced to 1000, there is greater variation in the firing rate and average membrane potential of the population. In the E-I network, a temporarily high number of spikes from the excitatory population leads to increased excitatory input 3 ms later and increased inhibitory input 3 ms after that. The resulting reduction in average membrane potential and firing rate is therefore exaggerated which produces an overall skew of these metrics. A population in MIIND can be thought of as an infinite number of trials of a single neuron or as an infinite number of neurons performing a single trial once. Because of this, a MIIND simulation is independent of the number of neurons in the population and cannot produce so-called finite-size effects. This can be a useful feature as it is not always as clear from a Monte Carlo simulation what behaviour stems from the finite size and what is a population-level effect.

Finally, the visualisation of the probability mass function in MIIND could prove to be a valuable educational tool for understanding the behaviour of neural populations under the influence of random spikes. In fact, any N-dimensional dynamical system under the influence of shot noise could be observed although this has not been attempted. It would be easy enough to plot points in a three-dimensional state-space for individually simulated neurons but points at the front of the distribution would obscure those at the back and in the centre. Producing a smooth enough distribution and picking an appropriate transparency value for each cell would require a population of millions of neurons.

The increased time to produce the probability mass function over direct simulation does not negate the usefulness of lower resolution grids to improve the simulation time as shown in Table 3.4. In particular, using a low-resolution grid can greatly improve workflow when designing or prototyping a new model. Building a model which performs as required involves multiple runs of the simulation as parameters are adjusted or when errors are identified. This is another reason why it is convenient that MIIND renders each population's probability mass function while the simulation is running. As shown in Fig. 3.11, viewing the probability mass function across all dimensions from any angle as the simulation progresses gives both insight into how the population behaves and any unexpected behaviour is quickly identified.

3.5.1 What is the theoretical output spike distribution of a population in MIIND?

Different populations in MIIND interact via their average firing rates. For each connection, the average firing rate of the source population is taken as the rate parameter to a Poisson distributed input to the target population. For a one-dimensional neuron model such as a leaky integrate-and-fire neuron for which incoming spikes cause an instantaneous jump in membrane potential, it is reasonable to assume that neurons in the population are pushed over threshold directly and only due to the Poisson distributed input suggesting that the output distribution should also be Poisson distributed. However, in higher-dimensional models with, for example, the addition of excitatory and inhibitory synaptic conductances, it becomes clear that neurons can move across threshold without direct influence from the Poisson input. If a sample of neurons is taken from the probability mass distribution at the beginning of a simulation, by definition, the probability that each sampled neuron is above threshold in a given time step is the probability mass which sits above threshold to be transferred to the reset potential. As the behaviour of all neurons is independent by virtue of being unconnected and homogeneous, the distribution of spiking neurons from the population at each time step can be considered binomial with p equal to the total probability mass above threshold. Using the average firing rate as the parameter to a Poisson input for each population is therefore a reasonable approximation. Models such as the E-I network which has self-connections and loop-connections invalidates the assumption of independence and further work is required to assess if using Poisson distributed outputs is appropriate under such circumstances.

3.5.2 Finite Size Populations

The main function of MIIND is to use the numerical population density approach to simulate population behaviour. However, a population of finite size can also be simulated which makes use of the transition matrix file and calculated jump transitions. This hybrid version of the algorithm is closer to direct simulation. A list of M grid coordinates is stored which represents the location in state-space of M individual neurons. At each time step, each coordinate is updated to one of the possible transition cells defined in the transition file with probability equal to the proportion of mass in that transition. To capture the non-deterministic dynamics, a Poisson distributed random number of spikes is sampled and the calculated jump transition is applied that many times. Again, the jump transition mass proportions are used as the probability for choosing the coordinate update with each jump. The average firing rate of the population is the number of neurons above threshold (which are then translated to the reset potential) divided by M. Currently direct connections between neurons are not implemented and instead, the average firing rate is used as the Poisson rate parameter applied to all neurons in the target population. Because the Poisson master equation is not required to solve the non-deterministic dynamics, this algorithm is much faster than the population density technique and approaches the speeds of simulations in GeNN although this has not been fully benchmarked. The two main reasons for using the finite size algorithm in MIIND are to further speed up prototyping of new models and to more easily eliminate finite-size effects.

3.5.3 Other potential models for study

The ability to easily simulate populations of three- and four-dimensional neuron models opens a world of possibilities for the population density approach. The Tsodyks-Markram synapse model (Tsodyks and Markram, 1997), for example, can be combined with a leaky integrate-and-fire neuron model to define a four-dimensional system. In the original work, the model was shown to support both rate coding between neurons and more precise spike timing based on the configuration of resource management in the synapse. For a large population, simulating the rate coding configuration makes more sense but MIIND could also be used to investigate the resilience of the spike timing configuration to noise. Booth and Rinzel, 1995 developed a two-compartment minimal motor neuron model. Each compartment requires two dimensions and MIIND would therefore be able to simulate a population of both compartments together. This model can reproduce the bi-stable behaviour of motor neurons such that a suitable incoming excitatory burst of spikes can shift the population to an up state where it remains even in the absence of further input. This is a candidate for identifying any finite-size effects and, in the presence of noise, estimating the amplitude and duration of the required excitatory and inhibitory bursts to switch states.

3.5.4 Limitations

The population density approach suffers from the so-called curse of dimensionality. With each additional time-dependent variable in the underlying neuron model, the number of cells in the grid is multiplied by the resolution of the new dimension. Not only does this produce an exponential increase in the number of cells for which the deterministic and non-deterministic dynamics must be solved, but the number of transitions per cell in the transition file also increases in most cases. The 50x50x50x50 transition file for the Hodgkin-Huxley model runs to nearly 1.5 Gb all of which must be loaded into graphics memory. There is still work to do to improve the memory management in MIIND but it is likely that a 5-dimensional transition matrix would not fit in the memory of current graphics hardware. In addition, generating the Hodgkin-Huxley transition file takes over 100 hours on the four CPU cores of a typical PC. This is a one-time preprocessing requirement that can be mitigated somewhat with high-performance computing systems but, again, for higher dimensional models the time required would become unfeasible.

In many cases, the number of cells in the grid that contain a non-zero amount of probability mass at any time during the simulation is much lower than the total number of cells. For higher dimensions, it would be possible to calculate the non-deterministic dynamics transitions required for a cell when probability mass is first transferred to it during the simulation. The simulation would be considerably slower at the beginning but would approach the original speeds as more cells are calculated. The memory requirements would only come from the cells involved in the probability mass function. This adaptation would still have an upper limit on the number of dimensions as the number of involved cells would still increase with greater dimensionality but it would be far from the exponential increase currently.

Another potential method for improving performance in both memory and computation speed would be to relax the requirement that all grid cells are the same size. In areas of state space where the dynamics are expected to follow a shallow curve (as opposed to the sharp turns in state space which can occur near unstable stationary points for example), larger cells could be defined. In order to preserve the benefit of equally sized cells when calculating the jump transition, the larger cells could be subdivided at simulation time and the deterministic dynamics transitions into the large cell could be linearly interpolated throughout. While not significantly affecting the computation time, the memory requirements would improve with the reduced number of transitions.

3.5.5 Conclusion

We have demonstrated for the first time, a numerical population density technique to simulate populations of N-dimensional neurons. Although models of higher than 5 dimensions are currently technologically out of reach, it is a significant achievement to produce the probability mass function of a population of 4-dimensional Hodgkin-Huxley neurons and to be able to visualise it in such a fashion. Implementing this technique in MIIND results in a very low barrier to entry for new users allowing them to define their desired neuron model in Python, automatically generate the required transition files and run the simulation without expert knowledge of the technique or any involved technical knowledge beyond some basic Python and XML. Although originally conceived as a technique to improve computational efficiency when simulating large populations of neurons, the population density technique cannot achieve the speeds of some other simulation methods. However, there are a number of benefits to using it, particularly in the areas of theoretical neuroscience and as a tool for analysis.

Acknowledgments

The authors wish to thank Frank van der Velde and Martin Perez-Guevara for their continued support of the MIIND project.

Data Availability Statement

The MIIND source code and installation packages are available as a github repository at `https://github.com/dekamps/miind`.

MIIND can be installed for use in Python using "pip install miind" on many Linux, MacOS, and Windows machines with python versions >= 3.6.

Documentation is available at `https://miind.readthedocs.io/`.

The leaky integrate-and-fire model, E-I network model, and Hodkin-Huxley model can be found in the examples/miind_nd_examples directory of the MIIND repository on github.

References

Barber, C Bradford, Dobkin, David P, and Huhdanpaa, Hannu (1996). “The quickhull algorithm for convex hulls”. In: *ACM Transactions on Mathematical Software (TOMS)* 22.4, pp. 469–483.

Bogacz, Rafal, Brown, Eric, Moehlis, Jeff, Holmes, Philip, and Cohen, Jonathan D (2006). “The physics of optimal decision making: a formal analysis of models of performance in two-alternative forced-choice tasks.” In: *Psychological review* 113.4, p. 700.

Booth, Victoria and Rinzel, John (1995). “A minimal, compartmental model for a dendritic origin of bistability of motoneuron firing patterns”. In: *Journal of computational neuroscience* 2.4, pp. 299–312.

Brette, Romain and Gerstner, Wulfram (2005). “Adaptive exponential integrate-and-fire model as an effective description of neuronal activity”. In: *Journal of neurophysiology* 94.5, pp. 3637–3642.

Brown, Kevin Q (1979). “Voronoi diagrams from convex hulls”. In: *Information processing letters* 9.5, pp. 223–228.

Brunel, Nicolas (2000). “Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons”. In: *Journal of computational neuroscience* 8.3, pp. 183–208.

Chizhov, Anton V, Campillo, Fabien, Desroches, Mathieu, Guillamon, Antoni, and Rodrigues, Serafim (2019). “Conductance-based refractory density approach for a population of bursting neurons”. In: *Bulletin of mathematical biology* 81.10, pp. 4124–4143.

Chizhov, Anton V and Graham, Lyle J (2007). “Population model of hippocampal pyramidal neurons, linking a refractory density approach to conductance-based neurons”. In: *Physical Review E* 75.1, p. 011924.

de Kamps, Marc (2006). “An analytic solution of the reentrant Poisson master equation and its application in the simulation of large groups of spiking neurons”. In: *The 2006 IEEE International Joint Conference on Neural Network Proceedings.* IEEE, pp. 102–109.

de Kamps, Marc, Lepperød, Mikkel, and Lai, Yi Ming (2019). “Computational geometry for modeling neural populations: From visualization to simulation”. In: *PLoS computational biology* 15.3, e1006729.

FitzHugh, Richard (1961). “Impulses and physiological states in theoretical models of nerve membrane”. In: *Biophysical journal* 1.6, p. 445.

Gewaltig, Marc-Oliver and Diesmann, Markus (2007). “NEST (NEural Simulation Tool)”. In: *Scholarpedia* 2.4, p. 1430.

Haiman, Mark (1991). “A simple and relatively efficient triangulation of the n-cube”. In: *Discrete & Computational Geometry* 6.3, pp. 287–289.

Hodgkin, Allan L and Huxley, Andrew F (1952). “The components of membrane conductance in the giant axon of Loligo”. In: *The Journal of physiology* 116.4, p. 473.

Izhikevich, Eugene M (2007). *Dynamical systems in neuroscience: The Geometry of Excitability and Bursting.* Cambridge, MA: MIT press.

Jansen, Ben H and Rit, Vincent G (1995). “Electroencephalogram and visual evoked potential generation in a mathematical model of coupled cortical columns”. In: *Biological cybernetics* 73.4, pp. 357–366.

Johannesma, Petrus Ignatius Marie (1969). “Stochastic neural activity: A theoretical investigation”. PhD thesis. Nijmegen: Faculteit der Wiskunde en Natuurwetenschappen.

Knight, Bruce W (1972). “Dynamics of encoding in a population of neurons”. In: *The Journal of general physiology* 59.6, pp. 734–766.

Knight, James C, Komissarov, Anton, and Nowotny, Thomas (2021). “PyGeNN: A Python Library for GPU-Enhanced Neural Networks”. In: *Frontiers in Neuroinformatics* 15.

Ly, Cheng and Tranchina, Daniel (2009). “Spike train statistics and dynamics with synaptic input from any renewal process: a population density approach”. In: *Neural computation* 21.2, pp. 360–396.

Mattia, Maurizio and Del Giudice, Paolo (2002). “Population dynamics of interacting spiking neurons”. In: *Physical Review E* 66.5, p. 051917.

Nagumo, Jinichi, Arimoto, Suguru, and Yoshizawa, Shuji (1962). “An active pulse transmission line simulating nerve axon”. In: *Proceedings of the IRE* 50.10, pp. 2061–2070.

Naud, Richard and Gerstner, Wulfram (2012). "Coding and decoding with adapting neurons: a population approach to the peri-stimulus time histogram". In.

Nykamp, Duane Q and Tranchina, Daniel (2000). "A population density approach that facilitates large-scale modeling of neural networks: Analysis and an application to orientation tuning". In: *Journal of computational neuroscience* 8.1, pp. 19–50. DOI: 10.1023/A:1008912914816. URL: https://doi.org/10.1023/A:1008912914816.

Omurtag, Ahmet, Knight, Bruce W, and Sirovich, Lawrence (2000). "On the simulation of large populations of neurons". In: *Journal of computational neuroscience* 8.1, pp. 51–63.

Osborne, H, Lai, YM, Lepperød, ME, Sichau, D, Deutz, L, and de Kamps, Marc (2021). "MIIND: A Model-Agnostic Simulator of Neural Populations". In: *Frontiers in Neuroinformatics* 15.

Potjans, Tobias C and Diesmann, Markus (2014). "The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model". In: *Cerebral cortex* 24.3, pp. 785–806.

Rangan, Aaditya V and Cai, David (2007). "Fast numerical methods for simulating large-scale integrate-and-fire neuronal networks". In: *Journal of computational neuroscience* 22.1, pp. 81–100.

Schmutz, Valentin, Gerstner, Wulfram, and Schwalger, Tilo (2020). "Mesoscopic population equations for spiking neural networks with synaptic short-term plasticity". In: *The Journal of Mathematical Neuroscience* 10.1, pp. 1–32.

Schwalger, Tilo and Chizhov, Anton V (2019). "Mind the last spike—firing rate models for mesoscopic populations of spiking neurons". In: *Current opinion in neurobiology* 58, pp. 155–166.

Schwalger, Tilo, Deger, Moritz, and Gerstner, Wulfram (2017). "Towards a theory of cortical columns: From spiking neurons to interacting neural populations of finite size". In: *PLoS computational biology* 13.4, e1005507.

Sirovich, L, Everson, R, Kaplan, E, Knight, BW, O'Brien, E, and Orbach, D (1996). "Modeling the functional organization of the visual cortex". In: *Physica D: Nonlinear Phenomena* 96.1-4, pp. 355–366.

Sukenik, Nirit, Vinogradov, Oleg, Weinreb, Eyal, Segal, Menahem, Levina, Anna, and Moses, Elisha (2021). “Neuronal circuits overcome imbalance in excitation and inhibition by adjusting connection numbers”. In: *Proceedings of the National Academy of Sciences* 118.12.

Tsodyks, Misha V and Markram, Henry (1997). “The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability”. In: *Proceedings of the national academy of sciences* 94.2, pp. 719–723.

Wilson, HR and Cowan, JD (1972). “Excitatory and inhibitory interactions in localized populations of model neurons”. In: *Biophysical journal* 12.1, pp. 1–24.

Yavuz, Esin, Turner, James, and Nowotny, Thomas (2016). “GeNN: a code generation framework for accelerated brain simulations”. In: *Scientific reports* 6.1, pp. 1–14.

York, Gareth James Richard, Osborne, Hugh, Sriya, Piyanee, Astill, Sarah, de Kamps, Marc, and Chakrabarty, Samit (2022). “The effect of limb position on a static knee extension task can be explained with a simple spinal cord circuit model”. In: *Journal of neurophysiology.*

Chapter 4

The Effect of Limb Position on a Static Knee Extension Task can be Explained with a Simple Spinal Cord Circuit Model (Paper C)

Abstract

The influence of proprioceptive feedback on muscle activity during isometric tasks is the subject of conflicting studies. We performed an isometric knee extension task experiment based on two common clinical tests for mobility and flexibility. The task was carried out at four pre-set angles of the knee and we recorded from five muscles for two different hip positions. We applied muscle synergy analysis using non-negative matrix factorisation on surface electromyograph recordings to identify patterns in the data which changed with internal knee angle, suggesting a link between proprioception and muscle activity. We hypothesised that such patterns arise from the way proprioceptive and cortical signals are integrated in neural circuits of the spinal cord. Using the MIIND neural simulation platform, we developed a computational model based on current understanding of spinal circuits with an adjustable afferent input. The model produces the same synergy trends as observed in the data, driven by changes in the afferent input. In order to match the activation patterns from each knee angle and position of the experiment, the model predicts the need for three distinct inputs: two to control a non-linear bias towards the extensors and against the flexors, and a further input to control additional inhibition of rectus femoris. The results show that proprioception may be involved in modulating muscle synergies encoded in cortical or spinal neural circuits.

Significance statement

The role of sensory feedback in motor control when limbs are held in a fixed position is disputed. We performed a novel experiment involving fixed position tasks based on two common clinical tests. We identified patterns of muscle activity during the tasks which changed with different leg positions and then inferred how sensory feedback might influence the observations. We developed a computational model which required three distinct inputs to reproduce the activity patterns observed experimentally. The model provides a neural explanation for how the activity patterns can be changed by sensory feedback.

4.1 Introduction

The execution of a motor task is achieved through the integration of simple movement commands which are modulated by sensory feedback from the periphery over time. The role of proprioceptive feedback in the recruitment of muscle fibres to counter load during a given task is well understood. However, its role in control of muscle activity, especially in a commonly tested static task involving a single joint, is still poorly understood. For example, in the study of a deafferented man (Rothwell et al., 1982), isometric control was shown to be impaired in some tasks. However, a report (Roh et al., 2012) on activation patterns in muscles of the upper arm during an isometric task, where the limb is restricted in place, showed no change when the arm position was altered. Previous studies of isometric knee extension tasks have recorded a change in muscle activation when the position of the limb is altered through a change in the knee or hip angles. However, the neural mechanism responsible for this change has not been confirmed (Haffajee et al., 1972; Onishi et al., 2002; Ema et al., 2017). Knee extensions are also used in a clinical test known as STREAM (Daley, 1997) to assess recovery of acute stroke patients and in a test to assess the flexibility of the hip flexors known as the Thomas test (Thomas, 1878). Though the protocols of these tests are well described, the potential effect of limb positions on the observed attributes of the muscle such as tension and strength, and associated mechanisms remain poorly examined.

In an isometric task that is performed at different limb positions, the following mechanical aspects should be considered. At different positions, all muscle lengths are potentially different but remain constant during the task. The maximum active force produced by a given muscle could also be different at each position due to the so-called force-length relationship of muscle fibres (Rack and Westbury, 1969). At different limb positions, the component of muscle force required to produce a torque will also change. Limb position can, therefore, be expected to affect the activity of Golgi-tendon organs and muscle spindles which are sensitive to force and muscle length respectively. However, identifying which afferents are responsible for an observed change in activity remains a complex task due to a lack of clarity around neural circuits in the spinal cord and higher areas of the central nervous system (CNS). Additional cutaneous afferents, particularly from skin stretch receptors or touch and pressure receptors could also be activated differently with changing limb position (Aimonetti et al., 2007) and this neural coding of position might be fed back to the motor units during such tasks.

Isometric tasks can help to disaggregate certain proprioceptive effects. For example, primary and secondary afferent pathways from muscle spindles have been shown to react to change in muscle length (B. Matthews, 1933; Lloyd, 1943). With the limb held in place, we can expect less influence from dynamic primary spindle afferents compared to an unconstrained limb. However, we cannot eliminate static primary and secondary spindle afferents because recruitment of muscles in the finger has been shown to increase their activity even in the absence of length change (Edin and Vallbo, 1990; Macefield et al., 1991) perhaps due to gamma motor neuron activation. Due to a reduced effect from primary spindle afferents, however, isometric tasks can be used to accentuate the activity of Ib afferents deriving from Golgi-tendon organs (Kistemaker et al., 2013). Ib afferents are sensitive to both the active and passive force production of a muscle and are not suppressed by the limb constraint in an isometric task.

Even knowing the source of proprioceptive activity in a task may not guarantee a positive identification of potential pathways and mechanisms involved. There are many possible targets to which afferent pathways have been shown to project. Spinal interneurons that were once thought to transmit signals from only one source such as Ia interneurons (Eccles et al., 1956) have since been shown to be supplied by Ib fibres as well (Jankowska and McCrea, 1983). Group II pathways project to interneurons and motor neurons beyond local agonists and antagonists and are incident on so-called Ib interneurons (Lundberg, 1979; Lundberg and Malmgren, 1988; McCrea, 1992). Beyond reflexes, it is clear that proprioceptive feedback is integrated in supraspinal neural circuits for maintaining balance and ongoing motor control tasks (Pruszynski and Scott, 2012; Safavynia and Ting, 2013). However, studies that do not directly interrogate afferent pathways can still provide a functional explanation for behaviour. In their pioneering modelling work, McCrea and Rybak, 2008 proposed a pattern generator model for locomotion in cats without explicit identification of the neurons involved.

Muscle synergy analysis is a tool for identifying common sources of activity from recordings of multiple muscles. The way in which the results of muscle synergy analysis change in response to limb position might give further insight into proprioceptive effects on muscle recruitment. Previous studies (Torres-Oviedo, Macpherson, et al., 2006; Roh et al., 2012) have not identified synergy changes in such cases. However, positive results may be forthcoming with a simpler isometric extension task. With respect to synergy analysis, a muscle synergy is a muscle recruitment pattern, often derived from electromyographic (EMG) recordings of multiple muscles and described in terms of time. Each muscle is given a weight value which indicates the amount that the recruitment pattern contributes to its activity. There is great variation in the way a motor task can be performed, even at a single joint. The use of muscle synergies by the CNS to alleviate the degrees of freedom problem is accepted but there is still disagreement about the mechanism of their recruitment (Grillner, 1985; Bizzi, Mussa-Ivaldi, et al., 1991; Tresch, Saltiel, d'Avella, et al., 2002; Harpaz et al., 2019; Desrochers et al., 2019). Similar synergies are reported across species, especially for routine repetitive tasks like locomotion in vertebrates (Yang et al., 2019). Often, synergy analysis is used to identify shared activity across multiple muscles during a task with the assumption that muscles that share similar activation patterns must have some common feature to produce them, be it mechanical or neural (Tresch, Saltiel, and E, 1999; Bizzi, Cheung, et al., 2008; Giszter, 2015). As well as providing insight directly, synergy patterns give a clear summary of the structure of experimental data and are therefore a good method for identifying changes and trends due to differing conditions even if the structure is not representative of a so-called motor module as suggested by Kutch and Valero-Cuevas, 2012.

A recent review of muscle synergy analysis (Cheung and Seki, 2021) recommends the use of neural models to reproduce the observed synergies to better identify the mechanism responsible for the results. Ideally, a neural model will also yield predictions to be later validated or otherwise. With this in mind, the aim of this study is to firstly, confirm that there is a modulation of muscle activity during a static knee extension task with changing limb position. The task is designed as a restricted form of the STREAM and Thomas tests with the knee constrained and the hip supported but not held. Secondly, we will show that the observed effect on muscle activation can be produced by well-understood spinal circuits with proprioceptive origins (Jankowska, Jukes, et al., 1967; Pratt and Jordan, 1987) by qualitatively matching the muscle synergy patterns from the task to those from a neural circuit model.

4.2 Materials and Methods

4.2.1 Ethics

The study was conducted according to the Declaration of Helsinki and all experimental protocols were approved by the University of Leeds Research Ethics Committee (reference number BIOSCI 16-004). Healthy subjects (n= 17, female = 8) with an age range of 18-30 (24.4± 2.57 years) were recruited to participate in this study. Exclusion criteria included previous knee or leg injuries, if participants had done exercise within 48 hours prior to testing, knee stiffness or self-reported pain, use of recreational or performance-enhancing drugs, ingested alcohol in the previous 24 hours or were unable to provide informed consent. Subjects provided informed written consent to the study, noting possible risks associated with the activity.

4.2.2 Data Collection

Surface electromyography (sEMG) was recorded from seven muscles of the subject's dominant leg; rectus femoris (RF), vastus lateralis (VL), vastus medialis (VM), semitendinosus (ST), biceps femoris (BF), medial gastrocnemius (MG) and tibialis anterior (TA) – of which the MG and TA were discarded from further analysis, due to low signal to noise ratio. Data analysis was therefore performed on the five remaining muscle recordings. The skin was prepared for electrodes with shaving, cleaning with alcohol wipes and then application of conductive electrode gel. Data were sampled at 2 kHz using wireless Delsys Trigno IM electrodes. Electrodes were placed on the muscle belly, positioned by landmarks as described by Rainoldi et al., 2004.

4.2.3 Experimental Protocol

Subjects were asked to lay supine on a standard medical examination bed. The dominant leg was held in a DonJoy TROM™ ADVANCE (DJO UK Ltd) locking knee brace. The brace is an adjustable rehabilitation device that surrounds the thigh and calf and can be locked at 10° intervals between 0° and 90°. When locked, the brace stops all extension or flexion of the knee. Subjects were then shown how to perform an isometric knee extension, keeping the foot against the bed. They were provided the resulting sEMG output recorded from RF as a feedback to help them learn the performance of the task. Subjects were asked to perform an isometric knee extension at maximal voluntary effort for five seconds, attempting to maximise RF activity. During the task, the activity of RF was monitored and the position of the limb was observed to ensure no movement occurred. This was repeated six times with a three minute rest between contractions. Using the brace, the dominant knee was fixed at one of four angles: 0°, 20°, 60°, and 90°. The angle of the knee was always measured against the hip joint and the bony prominence on the outside of the ankle. Data were collected in two different positions and sessions for each subject. A picture of the two positions is shown in Fig. 4.1. In position 1, the participant was supine with both legs flat against the bed. As the internal knee angle was increased from 0°, the dominant leg was flexed at the hip as well as the knee in the brace such that the foot was flat on the bed for 20°, 60°, and 90°. In position 2, the subject was moved down the bed such that the knee of the dominant leg was beyond the edge. The foot was supported by a chair and the contralateral leg was fully flexed at both the hip and knee so that the contralateral foot was flat against the bed. With increasing internal knee angle, the foot was lowered below the level of the bed but still supported by the chair. The position selected for each subject was randomised for their first session. In the second session, the subject performed the task in the other position.

4.2.4 Data Preprocessing

The sEMG time series were rectified then visually inspected and segmented into equal sections containing one burst each. Each single burst series was then low-pass filtered at 4Hz (second-order Butterworth filter) to produce a smooth output to encourage the muscle synergy process to capture differences such as maximal activity and baseline activity instead of more granular activation patterns. Non-negative matrix factorisation (NMF) was performed on each burst series across all sEMG channels. NMF was also performed after normalising each burst series to its maximum value.

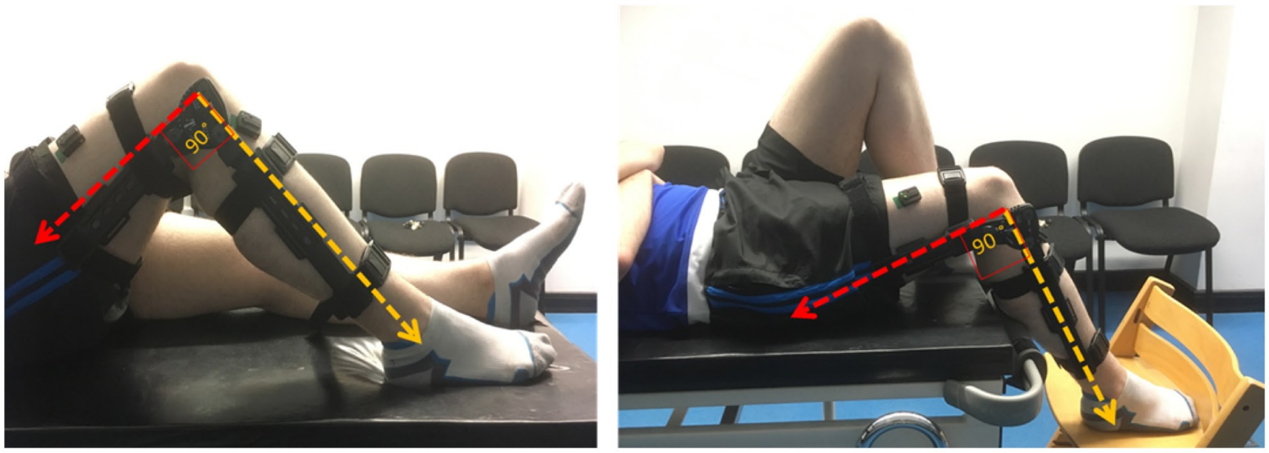


Figure 4.1: An image of the two positions of the experiment (position 1 on the left, position 2 on the right) and the leg brace used to constrain the knee.

4.2.5 Synergy Extraction

We used NMF to identify muscle synergies during the task. Information theory shows that the dimensionality reduction performed during NMF reflects latent structure in the data, which can be interpreted as muscle synergies (Lee and Seung, 2001). In muscle synergy analysis, a synergy refers to a component of the activity of one or more muscles. If the activity of two or more muscles contains a significant proportion of the same synergy, this may indicate that there is a "synergistic" relationship between the muscles. Here, the use of the term, synergy, refers to the components produced by the NMF process, not the relationship between muscles. NMF's chief advantage compared to other approaches is the constraint of non-negativity aligning with muscle activity i.e. muscle activation is never negative. It is, therefore, easier to interpret the resulting synergies. NMF is also more effective at identifying latent structure in the data when compared to other techniques such as principal component analysis (Ebied et al., 2018).

The five smoothed sEMG time series, for each muscle per burst, were combined into a matrix D of size $5 \times n$ where n is the length of the time series. We used iterative NMF decomposition algorithms (Tresch and Bizzi, 1999; Lee and Seung, 2001) to reduce D to a combination of two matrices, W and C such that,

$$D \approx WC \tag{4.1}$$

C is an $N \times n$ matrix where N is the chosen NMF rank factor, in this case, 2. Each row of C represents some structure in the time series similar to a principal component analysis component. W is a $5 \times N$ matrix which, when multiplied by C, approximates matrix D. Each column of W quantifies the amount that the corresponding row in C contributes to the original data in D (Lee and Seung, 2001; Donoho and Stodden, 2004; Berry et al., 2007; Torres-Oviedo and Ting, 2007). Each synergy, s, is represented by the corresponding column W_{*s} and row C_{s*}. We describe C_{s*} as the activation pattern of the synergy as it represents some underlying structure of the original sEMG time series. We refer to W_{*s} as the muscle contribution vector of the synergy as each component magnitude indicates the contribution of the synergy's activation pattern to the associated muscle activity.

Selection of rank factor is critical to achieving dimensionality reduction such that C has fewer rows than D. Rank factor was chosen consistent with previous literature (Tresch, Cheung, et al., 2006) such that rank factor was increased to the minimum required for the variance accounted for (VAF) by WC compared to D was greater than 90%. VAF was calculated for each synergy profile for both the individual muscle and for all muscles collectively. If VAF was below 90%, the resulting synergies were discarded. The iterative optimization algorithm used was initialised using singular value decomposition to reduce calculation time and to ensure a unique and reproducible result (Boutsidis and Gallopoulos, 2008). Each row in C and column in W was normalised to its maximum value. Cosine similarity analysis was used in a pairwise fashion to determine the similarity between subjects' synergy vectors and activation coefficients W_{*s} and C_{s*} (Rimini et al., 2017).

4.2.6 Neural Population Modelling

We aimed to create a neural population model such that applying NMF to the firing rate activity of the motor neuron populations would yield similar trends in activity and synergy patterns as those identified from the sEMG data. We did not attempt to reproduce simulated sEMG traces. Instead, we assumed that the cumulative activity of multiple motor units described by the average activity of distinct motor neuron populations would serve as a proxy for sEMG. When designing the model, we considered rate-based models which represent a population metric, for example, the average firing rate (Wilson and Cowan, 1972) or oscillation frequency (Kuramoto, 1991), abstracted from the underlying individual neurons. Rate-based models are suitable for reproducing firing rates in neural circuits, but there is no clear relationship with the state of the underlying neural substrate. Although not essential for this study, in light of more detailed spinal models used in the field where individual neurons are simulated (McCrea and Rybak, 2008), as well as future development of the modelling work, we are interested in a technique that retains a closer relationship with the state of the spiking neurons that comprise the neural circuit. Population Density Techniques (PDTs) provide such a balance: they retain information about the state of neurons in the circuits but calculate population-level aggregates directly.

4.2.7 Population Density Techniques

PDTs model neural circuits in terms of homogeneous populations of neurons. The individual neurons are described by a model, in this case, exponential integrate-and-fire. The model of an individual neuron is characterised by a so-called state space: the values that determine the state of individual spiking neurons. For a simple neuron model, this can be its membrane potential. For more complex models, variables such as the state of a synapse can be included. PDTs represent a population by a single density function that indicates how neurons are distributed across the neuron model's state space.

4.2.8 MIIND

MIIND is a neural simulator (de Kamps et al., 2008; Osborne et al., 2021) which implements a version of a PDT to simulate multiple interacting populations of neurons. It can provide a visual representation of the probability density function by displaying the density during simulation. Fig. 4.6B shows an example of this visual representation.

A network of populations can be built in MIIND using a simple XML style code format to list the individual populations and the connections between them. Populations in the network interact via their average firing rates, which are assumed to be Poisson distributed spike trains. For each connection, the firing rate of the source population becomes the average rate of the Poisson distributed input spikes to the destination population. The connections defined in the XML code have three parameters: the post-synaptic potential or instantaneous synaptic efficacy, the number of individual connections between source neurons and target neurons, and a delay which can be used to approximate time taken for spike propagation and synapse transmission.

Population name	Source population name	Post synaptic delta efficacy (mV)	Average number of incoming connections to each neuron	Connection delay time (ms)	Average firing rate where defined (Hz)
MN-RF	Extensor Interneurons	-0.052	20	2	
MN-VL	Extensor Interneurons	-0.052	20	2	
MN-VM	Extensor Interneurons	-0.052	20	2	
MN-ST	Extensor Interneurons	0.052	70	2	
MN-BF	Extensor Interneurons	0.052	70	2	
MN-RF	Flexor Interneurons	0.052	70	2	
MN-VL	Flexor Interneurons	0.052	70	2	
MN-VM	Flexor Interneurons	0.052	70	2	
MN-ST	Flexor Interneurons	-0.052	20	2	
MN-BF	Flexor Interneurons	-0.052	20	2	
MN-RF	InhibRF	-0.052	70	2	
Extensor Interneurons	Flexor Interneurons	-0.052	70	2	
Flexor Interneurons	Extensor Interneurons	-0.052	70	2	
‘ Extensor Interneurons	Background	0.1	100	0	300
Flexor Interneurons	Background	0.1	100	0	300
InhibRF	Background	0.1	100	0	300
MN-RF	Background	0.1	100	0	320
MN-VL	Background	0.1	100	0	320
MN-VM	Background	0.1	100	0	320
MN-ST	Background	0.1	100	0	320
MN-BF	Background	0.1	100	0	320
MN-RF	Cortical Drive	0.1	100	0	20*
MN-VL	Cortical Drive	0.1	100	0	20
MN-VM	Cortical Drive	0.1	100	0	20
MN-ST	Cortical Drive	0.1	100	0	20
MN-BF	Cortical Drive	0.1	100	0	20
Extensor Interneurons	Cortical Drive	0.1	100	0	20
Flexor Interneurons	Cortical Drive	0.1	100	0	20
Extensor Interneurons	senExtInt	0.1	100	0	0-15
Flexor Interneurons	senFlInt	0.1	100	0	0-150**
InhibRF	senInhRF	0.1	100	0	0-50**

** During the task, cortical drive input transitions from 0Hz to these values then back to 0Hz
*** During the task, afferent activity transitions from 0Hz to values in this range depending on the level of afferent feedback

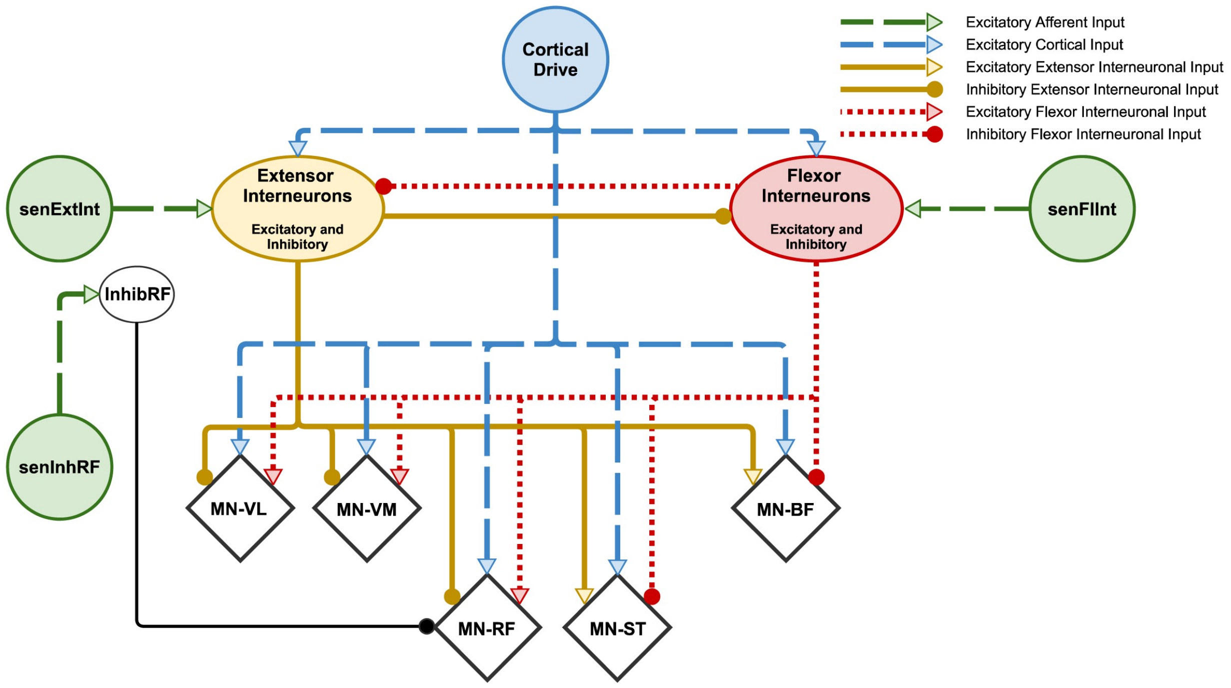
Table 4.1: Parameters relevant to each connection between populations and from inputs in the model. Values for input activity are provided in the form of an average firing rate.

4.2.9 The Spinal Circuit Model

We used MIIND to build a network of populations of exponential integrate-and-fire neurons according to the connectivity diagram in Fig. 4.2. Table 4.1 shows the connection parameters for all populations in the model. All populations use the same underlying neuron model as described in Equation 4.2.

$$\tau \frac{dv}{dt} = (v - v_{rest}) + \Delta_T e^{\frac{v - v_{thres}}{\Delta_T}} \tag{4.2}$$

Where v is the membrane potential, $v_{rest} = -70$ mV, $\Delta_T = 1.48$, $v_{thres} = -56$ mV, and $\tau = 3.3$ ms. The parameters were chosen so that populations could produce a wide range of average firing rates between 0 and 200 Hz to exhibit typical neuronal frequencies. We chose to use an exponential integrate-and-fire model in contrast to more commonly used Hodgkin-Huxley-style neurons. This is because the objective was not to reproduce the sEMG signals exactly, but to provide a concise explanation for the overall trends. We expected that any particular description of activation of ion channels (as in a Hodgkin-Huxley style model) would have no significant impact on the population level activity or synergy patterns in this task and would therefore dilute the power of the model.



Cortical Drive
Excitatory Afferent Input
Excitatory Cortical Input
Excitatory Extensor Interneuronal Input
Inhibitory Extensor Interneuronal Input
Excitatory Flexor Interneuronal Input
Inhibitory Flexor Interneuronal Input
senExtInt
Extensor Interneurons
Excitatory and Inhibitory
Flexor Interneurons
Excitatory and Inhibitory
senFlInt
InhibRF
senInhRF
MN-VL
MN-VM
MN-BF
MN-RF
MN-ST

Figure 4.2: Schematic of connections between simulated spinal populations. MN-VL (Vastus Lateralis), MN-VM (Vastus Medialis), MN-RF (Rectus Femoris), MN-ST (Semitendinosus) and MN-BF (Biceps Femoris) motor neuron populations are identified as diamonds although all populations consist of exponential integrate-and-fire neurons. The Extensor and Flexor Interneuron populations allow both outgoing excitatory and inhibitory connections. All populations receive a background level of input producing a baseline activity. Parameters for the network connectivity are provided in Table 4.1. The InhibRF population is used to offset the level of bias given to the MN-RF population. Afferent inputs senFlInt and senExtInt control the balance of input to the Flexor and Extensor Interneuron populations respectively which influences the agonist/antagonist bias. Afferent input senInhRF represents an additional input, activated to reproduce the change in activity of RF in position 2. Connections that exist in other models but which are not required to produce the observed synergies have been omitted. For example, direct afferent inputs to motor neuron populations. The relative strengths of each connection are not shown but can be found in the connectivity parameters (Table 4.1).

The main structure of the network consists of two neural populations, named “Extensor Interneurons” and “Flexor Interneurons”, connected together in a network with five motor neuron populations named MN-RF, MN-VL, MN-VM, MN-ST, and MN-BF for each respective muscle. The two interneuron populations are named for the group of populations of motor neurons that they inhibit. They also have excitatory connections to the remaining muscles. So for example, the Extensor Interneurons inhibit the knee extensors and excite the knee flexors. The interneuron populations represent combinations of excitatory and inhibitory neurons and therefore can project both kinds of connections to other populations in the network. This "network motif" is based on the idea of autogenic inhibition (Sherrington, 1909; Doss and Karpovich, 1965; Bigland-Ritchie, 1981; Pierrot-Deseilligny and Burke, 2005), a Ib afferent mechanism that inhibits the homonymous muscle. More recent work has shown that autogenic inhibition cannot be considered a local or self-contained reflex mechanism as force-dependent inhibition is part of a more distributed system (Nichols et al., 2016). However, we have chosen to use homonymous inhibition as a functional network pattern here. An alternative motif could have been based on reciprocal inhibition as observed with respect to the stretch reflex (Cooper, 1961; P. Matthews, 1969). In fact, the model does include reciprocal inhibition between the interneuron populations. However, from the perspective of the model, there is no functional difference between these two mechanisms unless a decision is made about the source of the afferent signals which are incident on the two interneuron populations. I.e, to simulate autogenic inhibition of one or more of the extensor motor neuron populations in the model, afferent input signals can be interpreted as Ib afferents originating from the extensor muscles and incident on the Extensor Interneuron population. To reproduce the stretch reflex on the same muscles, afferent signals can be interpreted as muscle spindle afferents from the extensor muscles and incident on the Flexor Interneuron population. These features, including the mutual inhibition between the two interneuron populations, also appear in the central pattern generator (CPG) model of McCrea and Rybak, 2008.

4.2.10 Cortical Drive

All supraspinal activity in the model comes from the cortical drive input and is responsible for the "contraction". It has been shown that corticospinal pathways are implicated in the control of more than just direct activation of motor neurons including gating and control of reflexes and pre-synaptic inhibition (Hultborn et al., 1987; Maier et al., 1998; Lemon and Griffiths, 2005; Pierrot-Deseilligny and Burke, 2005). However, for the purposes of this model, we assume that the pathways necessary to activate the motor neurons directly are open, that the contraction is produced by a small but significant increase in activity of these pathways (Shinoda et al., 1981), and that they do not change with limb position. There is a direct connection to all motor neuron populations. Each population receives the same level of activity from the cortical drive even though, in the task, the participant is asked to maximise the activity of RF. This is because direct comparison of the sEMG activity across muscles is unreliable. For example, if the level of sEMG activity of VM is twice that of RF, we cannot say that VM is twice as active because of differences such as the motor unit density, thickness and size of each muscle, and the distance between each muscle and the electrode. We should, therefore, expect that the output from the model does not match directly to the sEMG recordings across muscles but the same trends should still be observable. Cortical drive also projects to the extensor and flexor interneuron population to increase the excitability of the neurons so that they are more responsive to the afferent input signals. During the simulation, the input to the two interneuron populations and motor neuron populations begins at 0Hz before increasing to 20Hz over 1 second, then five seconds later, dropping back to 0Hz over 1 second. A frequency of 20Hz was chosen to match the beta frequency range commonly identified in voluntary motor control tasks (Salenius et al., 1997). The average firing rate of each of the five motor neuron populations was generated at a rate of 10 kHz (corresponding to the 0.1ms time step of the simulation) then sampled at 2ms intervals. NMF was performed on the resultant time series as described for the experimental recordings.

4.2.11 Afferent Inputs and the InhibRF Population

We hypothesised that the observed synergies are produced by the connectivity of the spinal neural network, chiefly the homonymous inhibition and heteronymous excitation. Furthermore, we expected that changes in afferent input would exaggerate or diminish the contribution vector values without significantly changing the activation patterns. External inputs could be made to any populations in the model including the motor neuron populations. In order to show that the trends in the activity patterns can be produced without changes to muscle-specific cortical control, we provided no afferent feedback above the level of the interneuron populations (cortical drive). Additionally, we have excluded direct external connections to the motor neuron populations. At the scale we have chosen to observe the muscle activity, identifying mono-synaptic vs oligo-synaptic connections is not feasible. Excitatory input to each motor neuron population can still be provided by either of the interneuron populations. As shown in Fig. 4.2, there are three separate afferent inputs in the model: senFlInt, senExtInt, and senInhRF. They were added to the model to specifically match the results of the experiment and so are discussed in the results section. Though not included in the original model, a further interneuron population, InhibRF, was added later to allow for control of the MN-RF motor neuron population separate from the MN-VM and MN-VL populations. Shevtsova et al., 2016 have previously used an additional inhibitory population of interneurons to reproduce behaviour of the bifunctional muscles, semitendinosus and rectus femoris, in a cat model. We believe this to be the first time such a technique has been applied to modelling in human studies. The function of the InhibRF population is discussed in further detail in the results section.

4.2.12 Statistical Analysis

All statistical analysis was performed in Python 3.6.2. Cosine similarity analysis was used to compare sEMG profiles, synergy activation patterns and muscle contribution vectors. Cosine similarity analysis is sensitive to differences in vectors that may have equal variation. Significance between sEMG burst time series was calculated using a two-sided T-test with $P < 0.05$ based on the mean value of the central four seconds of each burst corresponding to the majority of the "active phase" of the task. Significance between muscle contribution vector components was also calculated using a two-sided T-test with $P < 0.05$.

4.2.13 Code Accessibility

NMF analysis and cosine similarity analysis was performed using a custom-designed program in Python 3.6.2. MIIND is available at `https://github.com/dekamps/miind` and the model files and simulation results are accessible at `https://github.com/hugh-osborne/isotask`.

4.3 Results

4.3.1 Surface EMG activity changes with limb position

We recorded from 7 different muscles of the leg, but only 5 of these were used for further analysis of activity patterns as on examination the muscles TA and MG were always inactive, as expected due to the nature of the task. Fig. 4.3 shows the mean sEMG traces for each muscle, angle, and position. The mean value of the central four seconds of each sEMG trial was used to indicate the level of activity during the contraction. In position 1, there is a significant ($P < 0.05$) drop in the contraction activity of the quadriceps muscles RF, VL, and VM from 0° to 20°, and 20° to 60°. In position 2, the drop only occurs between 0° and 20° and is significantly greater than in position 1. For ST, there is a significant increase in the activity between 0° and 20° for both positions. Finally, at 20°, 60°, and 90°, RF has a significantly higher contraction activity in position 1 than position 2.

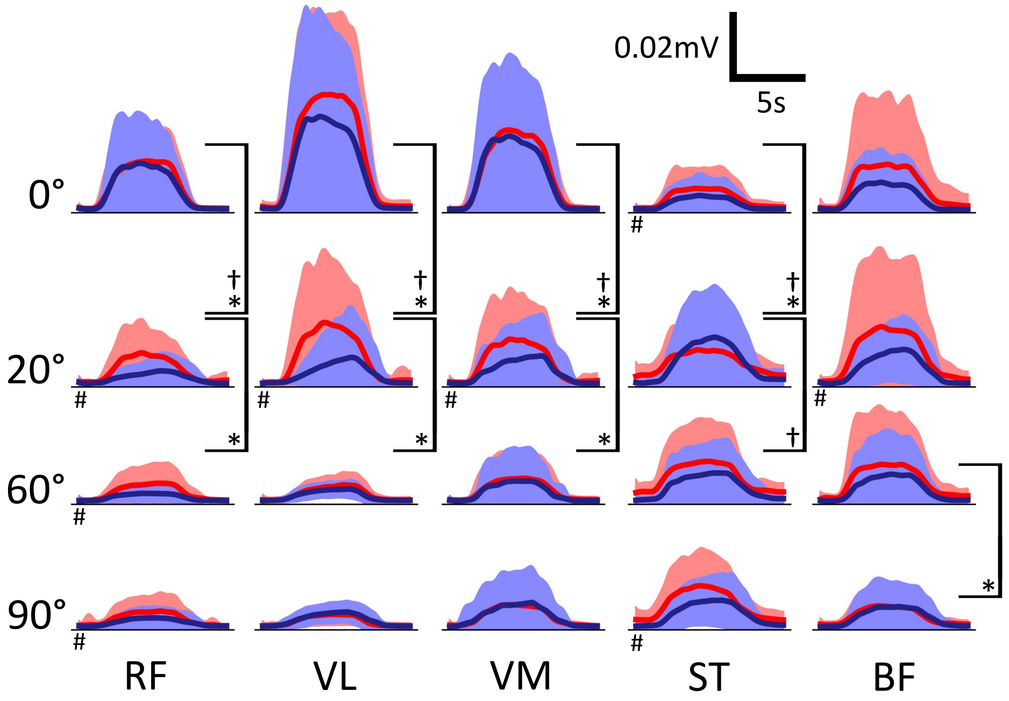
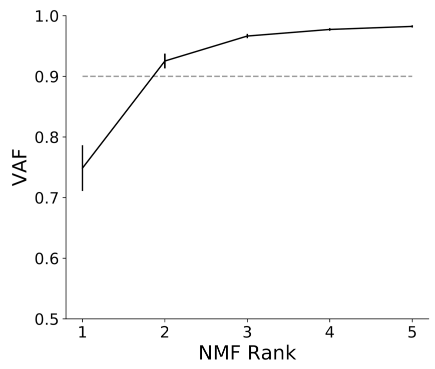


Figure 4.3: Mean sEMG traces for each muscle (columns), angle (rows), and position (bright red for position 1 and dark blue for position 2). Significance between plots with $P < 0.05$ were calculated based on a two sided T-test of the mean of the central four seconds of each sEMG trace to compare the activity of the contraction. $*$ represents significance between angles in position 1. † represents significance between angles in position 2. # represents significance between position 1 and position 2 for the same angle. N = 17 (M:9, F:8).

4.3.2 NMF identified two muscle synergies from the normalised sEMG activity

To identify synergies appropriate for experiment-model comparison, NMF was performed on the sEMG recordings, each normalised to its peak value, with a range of rank values. The appropriate rank to use was chosen as the number required to raise the VAF above 90% (Fig. 4.4). In this case, rank two raised VAF above this threshold. Although 90% is an arbitrary threshold, and there are other methods for choosing appropriate rank, patterns identified by three or more synergies were less consistent across participants. As described in section 4.2.4, each synergy consists of a column of matrix W with length five (one value per muscle) and a row of matrix C representing a time series describing some underlying structure of the original data. For each muscle, the corresponding component of W_{*s} multiplied by C_{s*}, gives the contribution of synergy, s, to that muscle's sEMG. Cosine similarity analysis was performed on the synergy rows and columns across participants for each position, synergy and angle. There is high correlation between synergy 1 results among the participants, regardless of position and internal knee angle (table in Fig. 4.4). Though not as high as synergy 1, there is also high correlation between participants for synergy 2. Despite some variation, there is a common pattern of muscle synergy recruitment across all participants.



	Synergy(s)				Angle
Position 1		0°	20°	60°	90°
Muscle Contribution Vector (W_{*s})	1	0.92	0.94	0.88	0.89
	2	0.77	0.66	0.69	0.58
Activation Pattern (C_{s*})	1	0.96	0.97	0.97	0.97
	2	0.66	0.75	0.73	0.69
Position 2					
Muscle Contribution Vector (W_{*s})	1	0.94	0.93	0.92	0.89
	2	0.67	0.65	0.62	0.70
Activation Pattern (C_{s*})	1	0.96	0.96	0.97	0.96
	2	0.68	0.74	0.73	0.76

Figure 4.4: Average VAF scree plot for rank one to five NMF dimensionality reduction across all angles and both positions of the static knee extension task. The 90% VAF threshold indicates that two is the appropriate rank to use and therefore the number of synergies to extract. Error bars show Standard Error of the Mean (SEM). In the table, synergy rows (activation patterns) and columns (contribution vectors as defined in section 4.2.4) were compared across all pairs of participants using cosine similarity analysis giving a value between 0 (uncorrelated) and 1 (Highly correlated). For both positions (activating or inactivating the contralateral hip flexors) and for all internal knee angles, there is high correlation between subjects indicating that, during the task, the same synergy patterns are being recruited by the majority of subjects. N = 17 (M:9, F:8).

4.3.3 Synergy 1 shows the coordinated recruitment of all five muscles

The NMF process generates, for each of the two synergies, a time series activation pattern and a vector of five values, one for each muscle. Fig. 4.5 shows the vector and time series of synergy 1 (A) and 2 (B) for both positions across different internal knee angles generated from the normalised sEMG recordings. The activation patterns (line plots) should be considered in conjunction with the five value muscle contribution vectors shown in the bar charts. Synergy 1 represents co-activation of all muscles and contributes to the majority of the observed sEMG activity. Because of this, the activation pattern closely matches the overall profile observed in the rectified and smoothed sEMG data (the transition from low to high to low activity during the contraction). The high muscle contribution values for all five muscles indicate that this activation pattern is present in all five sEMG recordings. Both the activation pattern and muscle contribution weights are well conserved across all angles, positions, and muscle groups.

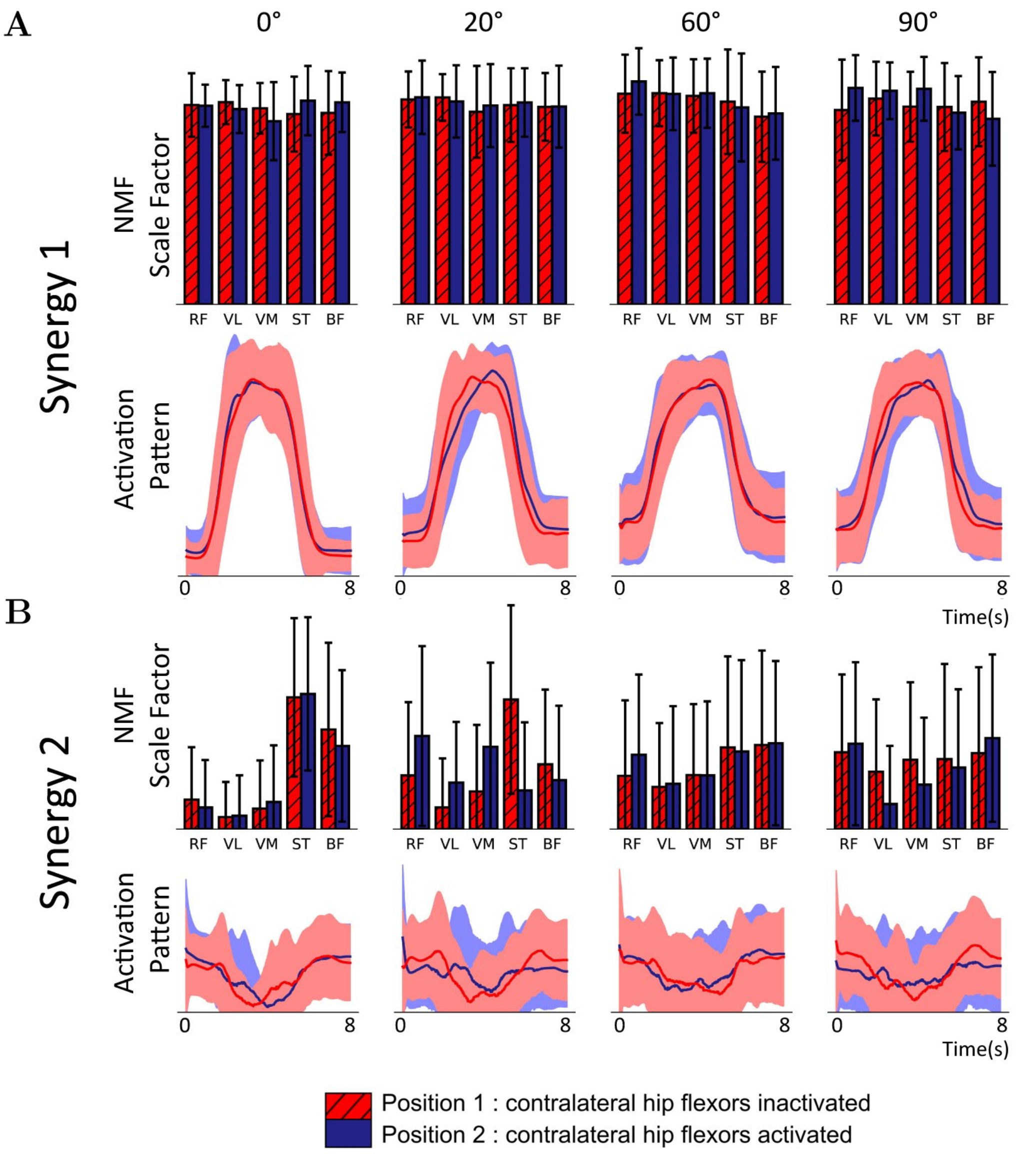


Figure 4.5: Muscle synergies extracted using rank two NMF from a static knee extension task at four internal angles of the knee (0°, 20°, 60°, and 90°) (N = 17, mixed gender, female = 8, age range of 18-30 (24.4±2.57 years). Subjects performed 6 contractions of 5s with the subject being asked to maximise rectus femoris activity. NMF was performed on the normalised sEMG of each subject's six contractions. The experiment was repeated across two positions inactivating (red values) or activating (blue) contralateral hip flexors. Line charts are activation patterns identified by NMF as underlying structure in the original sEMG time series. Filled areas show the standard deviation. Bar charts show the contribution of the associated activation pattern to the activity of each of the five muscles in arbitrary units. Error bars represent standard deviation. (A) Synergy 1 demonstrates the coordinated contraction across muscle groups in line with what is observed in the rectified and smoothed sEMG data. (B) Synergy 2 captures the inverse of the range of sEMG activity in each muscle. For both positions, at 0°, the antagonist muscles have significantly less activity than the agonists which results in high values for the antagonists. RF:rectus femoris; VL:vastus lateralis; VM:vastus medialis; ST:semitendinosus; BF:biceps femoris.

4.3.4 Changing the internal knee angle alters the contribution vector values of synergy 2

Normalising the sEMG data before performing NMF has the effect of setting the maximum activity level of each trial to 1 and scaling the remaining activity accordingly. This has the appearance of scaling the baseline activity of each trial which will be different across muscles, angles, and positions because of either, a difference in the maximal activity or a difference in the original non-normalised baseline. NMF chooses this feature for synergy 2 which explains the shape of the activation pattern. The contribution vector becomes an inverse measure of the difference between the baseline activity and the maximal activity and, as shown in Fig. 4.5B, the values change for different knee angles and positions.

In position 1 as the internal knee angle of the recorded leg is increased, the contribution vector value for ST reduces from a value far above those corresponding to the agonist muscles. A trend is less clear for the other antagonist muscle, BF, although the drop from 0° is still observable. The contribution vector values of RF, VL, and VM all increase with increasing angle. Overall, there is a flattening from the extreme differences at 0° as the internal angle approaches 90°. At 90°, magnitudes are similar for all muscles with neither an antagonist nor agonist bias. There is high variability in the contribution vectors at this angle and there appears to be no preference for any muscles in contrast to the lower angles.

4.3.5 Synergy 1 from the non-normalised sEMG suggests an increase in the contraction activity of ST

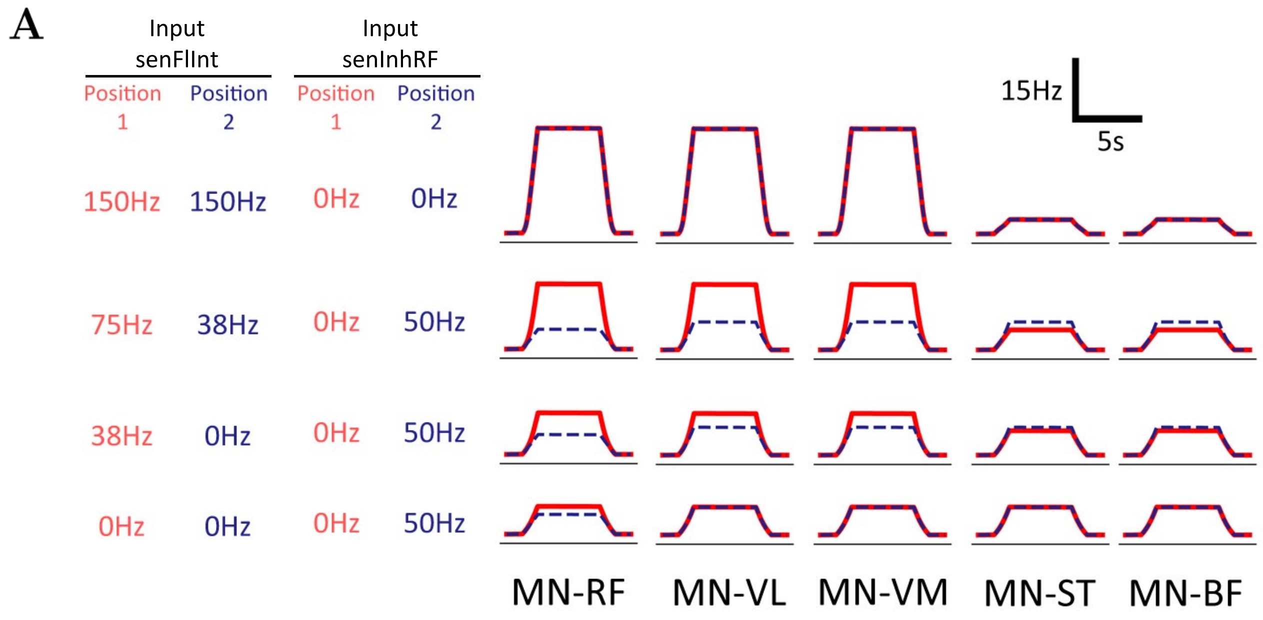
We also extracted two muscle synergies from the rectified and smoothed but not normalised sEMGs. As with the normalised version, the first synergy accounts for the change in activity due to the contraction of all muscles. The contribution vectors, however, now show similar trends observed in the rectified and smoothed sEMGs.

The second synergy has no common activation pattern between the angles or positions and this is true for any chosen rank. However, the contribution vector values for ST in position 1 at 20°, 60°, and 90° are higher than the other muscles which corresponds to the raised baseline activity visible in the smoothed sEMG recordings of Fig. 4.3. With this variance in ST isolated in another synergy, the first synergy shows a significant increase in the contribution vector values for ST for position 1 from 0° to

20°, and 20° to 60°.

4.3.6 The model with changes to the afferent input can reproduce trends in the sEMG recordings

The experimental observations above were used to design the spinal neural model presented in Fig. 4.2. The following results demonstrate the main behavioural features of the model and how they are influenced by the afferent inputs. During each MIIND simulation, the cortical drive input was changed from a low to high activity to simulate the contraction behaviour. The three afferent inputs could be changed to produce different effects on the output of the model. All populations produced average firing rates which were either passed to connected populations in the network or recorded for analysis. The activity of the five motor neuron populations, MN-RF, MN-VL, MN-VM, MN-ST and MN-BF, was analysed. The firing rate output from these populations is shown in Fig. 4.6A for a range of values of afferent inputs senFlInt and senExtInt which were chosen to produce firing rate outputs that qualitatively match the experimental sEMG traces. The blue dashed plots represent the firing rate output of each population chosen to match position 2. For example, the bottom row of dashed plots in the figure is produced by the model with senFlInt set to 0Hz and senInhRF set to 50Hz. The solid red plots are matched to position 1. The top row of traces is identical in both positions as both have senFlInt set to 150Hz and senInhRF set to 0Hz. Afferent input senExtInt was held constant at 0Hz as it is not required to simulate the observed trends. The need for senExtInt is discussed later. The output is much smoother than the sEMG recording data due to MIIND's simulation technique and the lack of many of the experimental sources of noise. There is undoubtedly a great deal more information available in the sEMG traces, but the model is designed only to explain the trends and significant observations from the experiment. In position 1, to produce a similar trend in the firing rate activities of the motor neuron populations to that of the sEMG traces with increasing internal knee angle, senFlInt must reduce non-linearly from 150Hz to 75Hz to 38Hz to 0Hz. For position 2, senFlInt must be reduced at a higher rate, immediately dropping from 150Hz to 38Hz to match the change from 0° to 20° in the experimental results. An alternative method for producing the trends without non-linear input is to use two or more functionally separate but linear afferent inputs. This is discussed in section 4.2.



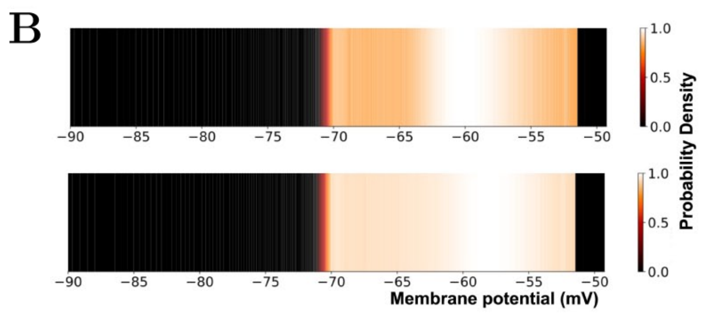


Figure 4.6: (A) Output firing rates of the five simulated motor neuron populations for different levels of afferent input. The significant differences observed in the sEMG data have been reproduced here with a non-linear reduction in activity of afferent input senFlInt and with a change in afferent input senInhRF between positions. Red solid lines represent approximations for position 1. Blue dashed lines represent approximations for position 2. (B) The probability density function of the MN-RF population in the model before input from cortical drive (upper) and during the contraction (lower). Colour brightness indicates the probability of a neuron in the population having the indicated membrane potential. The y-axis of the plots represents an arbitrary value for simple exponential integrate-and-fire neurons. A higher probability at the threshold of -51mV indicates a higher average firing rate for the population.

4.3.7 Afferent input senInhRF changes the contraction activity of RF to match positions 1 and 2

In order to generate the difference in the contraction activity of RF between positions 1 and 2, as observed in the rectified and smoothed sEMGs of Fig. 4.3, an additional inhibitory population was added to the model. The new population, InhibRF, inhibits only the MN-RF population and is facilitated by a separate afferent input, senInhRF. To match the reduced activity of RF in position 2, senInhRF is set slightly greater than in position 1 for 20°, 60°, and 90°, as shown in Fig. 4.6A.

4.3.8 Interpreting the MIIND Simulation Results

The heat plots in Fig. 4.6B show examples of the probability density functions produced by MIIND for each population in the network. As described in section 4.2.7, the density function shows the likelihood of finding a neuron from the population with a given membrane potential. The top density plot shows the state of the MN-RF population during the period before the contraction begins. The lower density plot shows the state when the input is maximal. In the lower density plot, there is a higher probability of finding neurons at the threshold (-51mV) indicating that the average firing rate of that population is higher. The population transitions to the top density once again after the cortical drive returns to zero. These transitions are also visible in the probability density functions of the other motor neuron populations due to the excitation from cortical drive. The behaviour of the cortical drive was designed to produce a similar activity pattern to the observed sEMG signals: an increase to a high level of activity followed by a decrease to rest.

4.3.9 The activation patterns and synergy 1 contribution vectors qualitatively match those derived from the experiment

In the same manner as the sEMG recordings, rank 2 NMF was performed on the normalised time series of average firing rates of the motor neuron populations in the model producing a five-value muscle contribution vector and time-series activation pattern for both synergies. The comparison was made with the NMF synergies derived from normalised sEMG. In the model, the average contraction activity of each motor neuron population is identical in the absence of afferent input. As explained in section 4.2.10, the sEMG contraction activity is different between muscles. By normalising the sEMG data, the differences between muscles are shifted from synergy 1 to synergy 2 where the trends due to changing internal knee angle can be seen more clearly although they are inverted. This is also true for normalised output from the model where the trends are only due to changes in afferent inputs. For each trial, the maximal activity value for the afferent input senFlInt was altered and the results were compared to those of the isometric task. Fig. 4.7 shows the results from the NMF process. For synergy 1 (Fig. 4.7A), the activation pattern matches the shape of the descending input pattern from the cortical drive input (5 seconds of maximal activity with a 1 second ramp up and down). The five muscle contribution values are all well above zero indicating that the activation pattern is a component in the activity of all the motor neuron populations. This is in good agreement with the synergy 1 pattern observed from the sEMG data. The shape of the synergy 2 activation pattern in Fig. 4.7B also appears qualitatively similar to that of the sEMG indicating that the same feature is being captured.

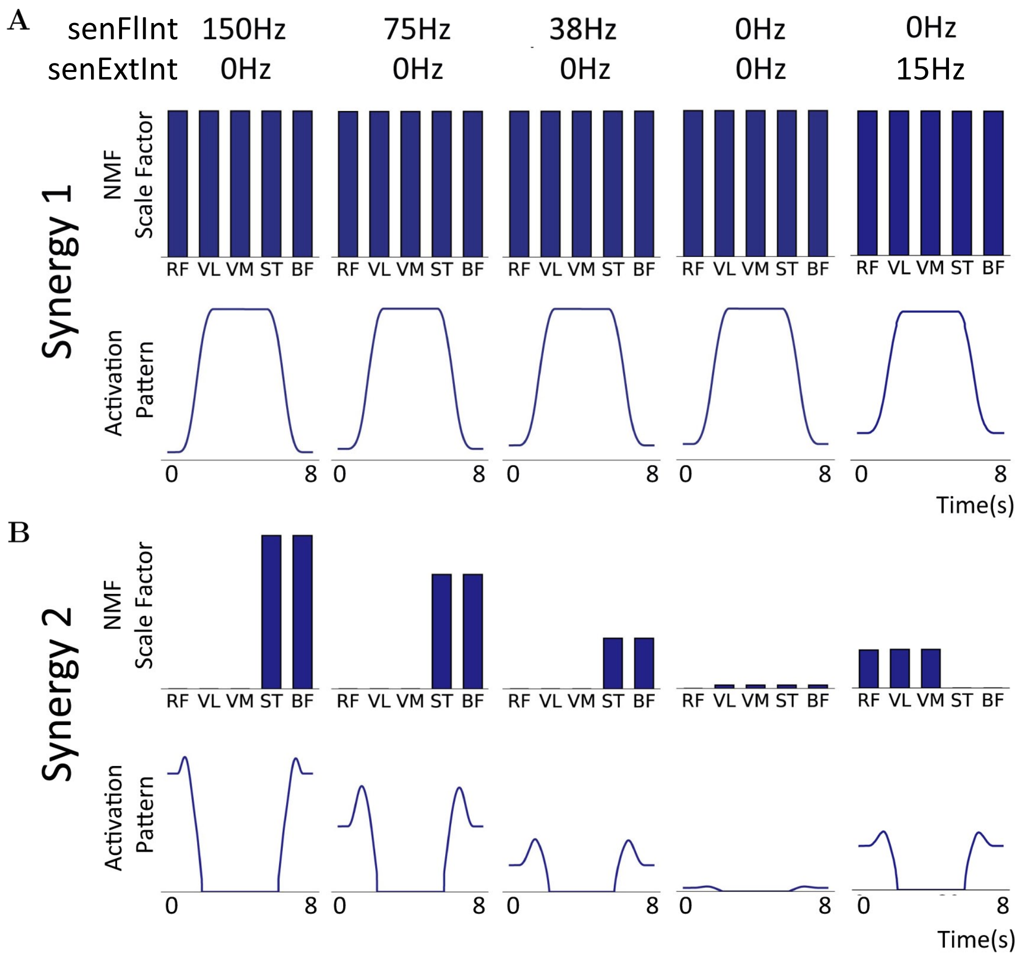


Figure 4.7: Muscle synergy features extracted using rank 2 NMF applied to the normalised average firing rates of the five motor neuron populations in the model for different levels of senFlInt and senExtInt (Fig. 4.2). As with the experimental results, line plots indicate the activation pattern for each synergy and bar charts indicate that pattern's contribution to each motor neuron population's activity. The contribution vector values are labelled with the muscle names which correspond to the 'MN-' motor neuron population names. (A) Synergy 1. The contribution vector is the same for all muscles across all levels of afferent activity. There is a small increase in the baseline of the activation pattern but the shape, caused by the cortical drive, appears at all angles. (B) Synergy 2. The high contribution vector values for the knee flexor populations are high when senFlInt is 150Hz. The values reduce with senFlInt and at 0Hz, there is effectively no synergy 2. When senExtInt is raised above senFlInt, the trend flips to an agonist bias.

4.3.10 Changing afferent inputs senFlInt or senExtInt reproduces the bias in synergy 2 between agonist and antagonist motor neuron populations

The degree to which the synergy 2 activation pattern contributes to each motor neuron population's activity changes with the different combinations of afferent inputs. Fig. 4.7B shows the trend in the synergy 2 contribution vector and activation pattern with decreasing senFlInt from left to right. When senExtInt is held at 0Hz and isenFlInt is greater than 0Hz, the synergy 2 activation pattern contributes to the knee flexor motor neuron populations significantly more than the extensors as was observed in the experimental results with lower internal knee angles. The rightmost column shows the synergies when senFlInt is held at 0Hz and senExtInt is raised. The bias in the contribution vector values of synergy 2 is flipped. An example of this flipped pattern can be seen in the synergy 2 contribution vector for position 2 at 20° (Fig. 4.5). In the model, when both senFlInt and senExtInt are raised above 0Hz, the bias is dependent on which input has higher activity. An additional excitation from senFlInt over senExtInt causes an imbalance in activity between the extensor interneuron population and flexor interneuron population. The resultant higher firing rate of the flexor interneuron population causes additional inhibition of MN-ST and MN-BF and excitation of MN-RF, MN-VL and MN-VM. Therefore, during the contraction, the extensor motor neuron populations have a higher maximal firing rate than the flexor populations. The same pattern can be seen in the sEMG traces of Fig. 4.3 at 0°. This mechanism is how bias in the synergy pattern to either the extensors or flexors is controlled. As the difference between afferent inputs senFlInt and senExtInt is reduced, the contribution of this pattern lowers until it is eliminated across all five populations.

4.4 Discussion

The major findings of this study show how muscle recruitment in static tasks changes with limb position. The observed trend in the recorded quadriceps, a non-linear increase in maximal activity as the internal knee angle approaches full extension, can be reproduced by a neural population model integrating afferent feedback. A separate effect on the maximal activity of RF was observed between the two positions.

4.4.1 Differences between the model synergies and experimental synergies

The synergy 2 contribution vectors from the sEMG show a decrease in the values for the knee flexor muscles and an increase in the values for the extensor muscles between 0° and 90°. 90° shows a common mean value for all muscles with high variability (Fig. 4.5). This is possibly because, in each trial at 90°, it was equally likely that NMF would find one or more muscles with a lower range of activity than the others. This is backed up by the mean sEMG data which show a common mean level of activity across muscles at 90°. In the model, such variation does not occur. Therefore, the vector values of synergy 2 for the extensor motor neuron populations remain at zero and, as senFlInt reduces to zero, so do the vector values for the flexor motor neuron populations.

4.4.2 What is the source of the afferent input in the model?

As mentioned, the model is somewhat agnostic about the source of the afferent inputs because there are many possible sources and mechanisms which could be inferred from the data. Afferent input senFlInt in the model could be produced by some cutaneous signal, perhaps pressure from the brace or the skin touching the bed. This cannot be ruled out. However, thinking about the mechanical aspects of the task, the participant works to maintain the leg position while "maximising activity of RF". So at all angles and positions, the aim is to keep the muscle-evoked forces in balance so that the leg does not lift off the bed. A neural mechanism for achieving this could be cortical, spinal, or distributed but it would still require proprioceptive feedback. We know of two well-understood spinal reflex mechanisms which can perform this function: the stretch reflex and autogenic inhibition and these were used to build the model.

Making the assumption that proprioception is responsible for the trends observed in the data, we can eliminate the dynamic Ia stretch response from the list of possible sources. During the contraction task, there are no changes in muscle length that would elicit the dynamic response (P. Matthews, 1969; Edin and Vallbo, 1990). It is possible, however, that the static response of both primary and secondary spindle afferents might change with internal knee angle. In that case, we would expect higher afferent activity from muscles that are stretched further. In the design of the model, the decision was made to exclude excitatory mono-synaptic afferent connections to homonymous motor neurons. If these had been included, the model would have produced the opposite trend (increased activation of the quadriceps with increasing internal knee angle) to what was observed. Instead of or in addition to muscle length, if tendon force is the source of the proprioceptive signal, we would again expect higher afferent activity from stretched muscle due to the force-length relationship (Rack and Westbury, 1969). In other isometric extension tasks, it might be expected that the passive force-length relationship would be overpowered by the response to active force from the muscle. However, in this task, the muscle activity appears to be low across all muscles including RF compared to sEMG recordings of the same muscles in similar tasks from other studies (Saito and Akima, 2015; Ema et al., 2017). Furthermore, at 0° and 20°, the hamstrings will be significantly stretched, enhancing the effect of passive force.

At 0°, is the greater activity in the quadriceps compared to the hamstrings a result of increased facilitation or decreased inhibition? We have made the simplest assumption, that there is additional activation of the quadriceps at 0° rather than supposing a common inhibitory signal across all muscles and angles which is itself inhibited in the quadriceps at 0°. This final assumption, combined with that of a proprioceptive signal which gets stronger with increasing muscle length is supported by the model we have presented here. As the internal knee angle decreases, the length of the hamstrings increases which produces a stronger proprioceptive signal exciting the quadriceps and inhibiting the hamstrings. Though our model implicates the well-known Ib autogenic inhibitory circuit, it is the heteronymous facilitation that is strongest. We only see increasing inhibition with decreasing angle in ST, not BF, so we cannot be sure that another pathway and possibly another proprioceptive source is responsible.

With the above considerations, it is difficult to deduce the source of afferent input senInhRF which affects the contraction activity of RF differently in the two positions. It could be the case that the hip is slightly lowered in position 2, due to the flexion of the contralateral knee and hip. This was not measured in the experiment but could affect the length of RF in the two positions. We might expect a different hip position to also affect ST and there is a visible difference in the contraction activities between the two positions but we cannot show significance at the same angles. We also cannot rule out a heteronymous interaction from a muscle that changed in the two positions but was not recorded.

In choosing values of activity for senFlInt, we required a non-linear scale to match the trend in the sEMG and synergy 2 vector values from the experiment. Fig. 4.6A and Fig. 4.7 show how senFlInt must reduce exponentially in order to match the equivalent trends with increasing knee angle from the experiment. Though the change in internal knee angle in the experiment is itself non-linear (the 20° angle would need to be replaced with 30°) it still appears likely that the afferent input does not scale proportionally with the change in internal knee angle. An alternative possibility is that there are two or more functionally separate inputs that sum to produce a non-linear effect on the resultant activity. For example, it is possible that one input might produce a common high contraction activity in both positions at 0° but not 20°, and another may cause higher activity at 20° only in position 1. However, a non-linear signal which applies to both positions is the simpler explanation which is why it was chosen for the model.

4.4.3 What are the muscle synergies?

Performing NMF on the normalised sEMG data yielded two separate activation patterns which were common across all internal knee angles. Normalising the data beforehand made for an easier comparison of the contribution vectors with the synergies from the model output. An important distinction between these results and the results of other synergy analysis studies (Saltiel et al., 2001; Dominici et al., 2011) is that the contribution vector values are not disjoint across synergies here. ST and BF have high contribution vector values in both synergies at 0° indicating that both activation patterns contribute to the overall sEMG activity of the hamstrings. However, when the sEMG recordings are not normalised, the common activation pattern of the second synergy disappears. This casts doubt that synergy 2 is representative of a motor module: a common activation pattern produced by multiple muscles to alleviate the degrees of freedom problem. Synergies are often derived from much more than 5 muscles and for more complex tasks such as locomotion. More muscles and greater task complexity increase the need for simplifying motor modules. However, even in this simple task, there is redundancy in the level of activation of the opposing agonist and antagonist muscles. Synergy 1 cannot, therefore, be similarly discounted as a motor module. Some studies (Milosevic et al., 2017; Desrochers et al., 2019; Alessandro et al., 2020) have found synergies or motor modules encoded in the spinal cord. The cortical involvement in synergy encoding also cannot be ruled out (Delis et al., 2018; Harpaz et al., 2019) but our results suggest that motor modules might be modulated by proprioceptive input at the level of the spinal cord. This is in agreement with recent work by Cheung, d'Avella, et al., 2005 and Santuz et al., 2019 who showed significant differences in synergy activation patterns and contribution vectors in the absence of key proprioceptive signals.

4.4.4 Differences to other isometric knee extension tasks

In the experiment, the knee is held in a brace and the leg is supported in both positions. However, the brace itself is not constrained and so the hip can be flexed. While the instruction is explicitly given to maximise the activity of RF, with visual feedback provided from the RF sEMG, there is an implicit instruction for the foot to be kept on the bed, i.e for the hip angle to be held constant. These instructions are in conflict but adherence to the second instruction was monitored during the experiment and an increase in the activity of RF during the contraction indicated that, if not maximal, some effort was being made to contract RF. The results show a lower level of muscle activity and a decrease in maximal sEMG activity with increasing internal knee angle which is the opposite trend to other isometric knee extension studies (Haffajee et al., 1972; Zabik and Dawson, 1996; Pincivero et al., 2004). Additionally, all recorded quadriceps and hamstring muscles were engaged instead of just the quadriceps, most likely to keep the hip at a constant angle.

The experimental protocol was designed to mimic the limb positions formed in the STREAM and Thomas tests. The benefit of recording from two positions was that we could identify differences in muscle activation for the same internal knee angles but different hip angles. In the Thomas test, the patient lies supine with their contralateral knee and hip maximally flexed and the foot of the observed leg supported by the bed. The observed knee and hip are relaxed and the resting knee angle is recorded. The modified Thomas test is performed similarly but with the observed leg unsupported over the edge of the bed. Our findings show that there is a difference in quadriceps activation when the internal knee angle is close to $0°$ and when it is closer to $90°$ so a different afferent response should be expected between the test and its modified version. Our findings further suggest that these tests, if combined with sEMG recording, could be used to monitor changes in propriospinal pathways.

Multiple studies have shown that Parkinson's disease affects proprioception, reducing the ability to accurately sense limb position (Adamovich et al., 2001; Mongeon et al., 2009; Artigas et al., 2016). While there appears to be little effect from Parkinson's disease on the muscle spindles or afferent pathways at the level of the spinal cord, the source of the effect in the brain remains unknown. A candidate area in the brain is the Supplementary Motor Area (Jacobs and Horak, 2006). By combining a neural model of this area with the spinal model proposed here, a better prediction about the influence of Parkinson's Disease on proprioception could be made in the future.

4.4.5 The use of MIIND

Instead of using the traditional technique of direct simulation of individual neurons, we have demonstrated the use of the MIIND simulation package, a software environment allowing easy modelling of populations of neurons. MIIND requires only the definition of connectivity at the population level, making it easy to set up and adjust a population network during development. Parameter tweaking is an inevitable part of the modelling process requiring cycles of adjustment followed by simulation. Reducing the need for adjustments to the neuron model itself was one reason why we used the simple exponential integrate-and-fire instead of a more complex Hodgkin-Huxley style neuron. We were able to reproduce the desired synergy patterns without the need for such complexity. While building the network model we experimented with different connection configurations between populations. MIIND's XML style code, used to describe the network, made it simple to add, remove or adjust connections, as well as to add further populations for the RF and ST bias. For one-dimensional neuron models, MIIND can simulate a population network with much greater speed than direct methods and this allowed simulations to be run on a local machine without the need for high-performance computing, significantly improving the turnaround time between changing and testing the model. From our experience here, we advocate the use of simple neuron models where appropriate, i.e. reduce the dimensionality of the neural model as far as possible. First, this increases simulation speed and second, this forces thinking about which are the essential neuronal mechanisms before simulation starts.

One way to evaluate the success of a model is to consider how it might be integrated into larger models to answer different research questions. CPG models are constructed from mutually inhibiting populations of bursting neurons to produce an oscillating pattern of activity for driving rhythmic behaviours in many species and areas of the body. The circuitry of motor neurons and interneurons below the two-layer CPG model for driving fictive locomotion in cats (McCrea and Rybak, 2008) has many similarities to the model proposed here. Both include mutually inhibiting populations of interneurons, with a proprioceptive input. The use of separate inhibitory populations for controlling bifunctional muscles was also first demonstrated by Shevtsova et al., 2016. Integrating our model would require a decision about whether the cortical drive should be mediated by the CPG or if it should bypass it. Answering this question would give insight into how voluntary movements and cycling (CPG-controlled) movements are performed by the same set of neural circuits.

4.4.6 Conclusion

In conclusion, there is a level of disagreement in the literature about the effect of proprioception on muscle activity in isometric tasks and about the effect of proprioception on synergies derived from recorded muscle activity. The synergy analysis and analysis of the sEMG data from our experiment clearly shows that there is an effect caused by a change in limb position which we attribute to proprioceptive signals. In a static knee extension task, as the internal knee angle approaches 0° at maximal extension, the activity of the agonist muscles during contraction increases in a non-linear fashion. In two different positions, but with the same internal knee angle, rectus femoris displays an increased level of activity during contraction with the contralateral leg straightened. By performing muscle synergy analysis on the sEMG data, we identified a possible additional trend in semitendinosus and used the generated synergy patterns to design a neural model. We more easily compared the output of the model, which produces average motor neuron population firing rates, with the sEMG results by normalising before performing NMF. Though this yielded two clear synergy patterns, the second is unlikely to be representative of a so-called motor module used to solve the degrees of freedom problem. When building neural models for comparison with sEMG, we recommend the use of simple neuron models such as exponential integrate-and-fire where ion channel dynamics are not required to explain observations. Finally, the model we have demonstrated here should be integrated into larger models of motor control to add the observed influence of proprioception at the spinal cord level.

References

Adamovich, SV, Berkinblit, MB, Hening, W, Sage, J, and Poizner, H (2001). “The interaction of visual and proprioceptive inputs in pointing to actual and remembered targets in Parkinson’s disease”. In: *Neuroscience* 104.4, pp. 1027–1041.

Aimonetti, JM, Hospod, V, Roll, JP, and Ribot-Ciscar, E (2007). “Cutaneous afferents provide a neuronal population vector that encodes the orientation of human ankle movements”. In: *The Journal of physiology* 580.2, pp. 649–658.

Alessandro, C, Barroso, FO, Prashara, A, Tentler, DP, Yeh, HY, and Tresch, MC (2020). “Coordination amongst quadriceps muscles suggests neural regulation of internal joint stresses, not simplification of task performance”. In: *Proceedings of the National Academy of Sciences* 117.14, pp. 8135–8142.

Artigas, NR, Eltz, GD, Severo do Pinho, A, Torman, VBL, Hilbig, A, and Rieder, CRM (2016). “Evaluation of knee proprioception and factors related to Parkinson’s disease”. In: *Neuroscience journal* 2016.

Berry, MW, Browne, M, Langville, AN, Pauca, VP, and Plemmons, RJ (2007). “Algorithms and applications for approximate nonnegative matrix factorization”. In: *Computational statistics & data analysis* 52.1, pp. 155–173.

Bigland-Ritchie, B (1981). “EMG/force relations and fatigue of human voluntary contractions”. In: *Exercise and sport sciences reviews* 9.1, pp. 75–118.

Bizzi, E, Cheung, VCK, d’Avella, A, Saltiel, P, and Tresch, M (2008). “Combining modules for movement”. In: *Brain research reviews* 57.1, pp. 125–133.

Bizzi, E, Mussa-Ivaldi, FA, and Giszter, S (1991). “Computations underlying the execution of movement: a biological perspective”. In: *Science* 253.5017, pp. 287–291.

Boutsidis, C and Gallopoulos, E (2008). “SVD based initialization: A head start for nonnegative matrix factorization”. In: *Pattern Recognition* 41.4, pp. 1350–1362.

Cheung, VCK, d'Avella, A, Tresch, MC, and E, Bizzi (2005). "Central and sensory contributions to the activation and organization of muscle synergies during natural motor behaviors". In: *Journal of Neuroscience* 25.27, pp. 6419–6434.

Cheung, VCK and Seki, K (2021). "Approaches to revealing the neural basis of muscle synergies: a review and a critique". In: *Journal of Neurophysiology* 125.5, pp. 1580–1597.

Cooper, S (1961). "The responses of the primary and secondary endings of muscle spindles with intact motor innervation during applied stretch". In: *Quarterly Journal of Experimental Physiology and Cognate Medical Sciences: Translation and Integration* 46.4, pp. 389–398.

Daley, K (1997). "The Stroke Rehabilitation Assessment of Movement (STREAM): refining and validating the content". In: *Physiother Can* 49, pp. 269–278.

de Kamps, Marc, Baier, V, Drever, J, Dietz, M, Mösenlechner, L, and Van der Velde, F (2008). "The state of MIIND". In: *Neural Networks* 21.8, pp. 1164–1181.

Delis, I, Hilt, PM, Pozzo, T, Panzeri, S, and Berret, B (2018). "Deciphering the functional role of spatial and temporal muscle synergies in whole-body movements". In: *Scientific reports* 8.1, pp. 1–17.

Desrochers, E, Harnie, J, Doelman, A, Hurteau, MF, and Frigon, A (2019). "Spinal control of muscle synergies for adult mammalian locomotion". In: *The Journal of physiology* 597.1, pp. 333–350.

Dominici, N, Ivanenko, YP, Cappellini, G, d'Avella, A, Mondi, V, Cicchese, M, Fabiano, A, Silei, T, di Paolo, A, Giannini, C, et al. (2011). "Locomotor primitives in newborn babies and their development". In: *Science* 334.6058, pp. 997–999.

Donoho, D and Stodden, V (2004). "When Does Non-Negative Matrix Factorization Give a Correct Decomposition into Parts?" In: *Advances in Neural Information Processing Systems 16*. Ed. by S. Thrun, L. K. Saul, and B. Schölkopf. MIT Press, pp. 1141–1148.

Doss, WS and Karpovich, PV (1965). "A comparison of concentric, eccentric, and isometric strength of elbow flexors". In: *Journal of Applied Physiology* 20.2, pp. 351–353.

Ebied, A, Kinney-Lang, E, Spyrou, L, and Escudero, J (2018). "Evaluation of matrix factorisation approaches for muscle synergy extraction". In: *Medical engineering & physics* 57, pp. 51–60.

Eccles, JC, Fatt, P, and Landgren, S (1956). “Central pathway for direct inhibitory action of impulses in largest afferent nerve fibres to muscle”. In: *Journal of neurophysiology* 19.1, pp. 75–98.

Edin, BB and Vallbo, AB (1990). “Muscle afferent responses to isometric contractions and relaxations in humans”. In: *Journal of neurophysiology* 63.6, pp. 1307–1313.

Ema, R, Wakahara, T, and Kawakami, Y (2017). “Effect of hip joint angle on concentric knee extension torque”. In: *Journal of Electromyography and Kinesiology* 37, pp. 141–146.

Giszter, SF (2015). “Motor primitives—new data and future questions”. In: *Current opinion in neurobiology* 33, pp. 156–165.

Grillner, S (1985). “Neurobiological bases of rhythmic motor acts in vertebrates”. In: *Science* 228.4696, pp. 143–149.

Haffajee, D, Moritz, U, and Svantesson, GJAOS (1972). “Isometric knee extension strength as a function of joint angle, muscle length and motor unit activity”. In: *Acta Orthopaedica Scandinavica* 43.2, pp. 138–147.

Harpaz, NK, Ungarish, D, Hatsopoulos, NG, and Flash, T (2019). “Movement decomposition in the primary motor cortex”. In: *Cerebral cortex* 29.4, pp. 1619–1633.

Hultborn, H, Meunier, S, Pierrot-Deseilligny, E, and Shindo, M (1987). “Changes in presynaptic inhibition of Ia fibres at the onset of voluntary contraction in man.” In: *The Journal of physiology* 389.1, pp. 757–772.

Jacobs, JV and Horak, FB (2006). “Abnormal proprioceptive-motor integration contributes to hypometric postural responses of subjects with Parkinson’s disease”. In: *Neuroscience* 141.2, pp. 999–1009.

Jankowska, E, Jukes, MGM, Lund, S, and Lundberg, A (1967). “The effect of DOPA on the spinal cord 5. Reciprocal organization of pathways transmitting excitatory action to alpha motoneurones of flexors and extensors”. In: *Acta Physiologica Scandinavica* 70.3-4, pp. 369–388.

Jankowska, E and McCrea, DA (1983). “Shared reflex pathways from Ib tendon organ afferents and Ia muscle spindle afferents in the cat.” In: *The Journal of Physiology* 338.1, pp. 99–111.

Kistemaker, DA, Knoek van Soest, AJ, Wong, JD, Kurtzer, I, and Gribble, PL (2013). "Control of position and movement is simplified by combined muscle spindle and Golgi tendon organ feedback". In: *Journal of neurophysiology* 109.4, pp. 1126–1139.

Kuramoto, Y (1991). "Collective synchronization of pulse-coupled oscillators and excitable units". In: *Physica D: Nonlinear Phenomena* 50.1, pp. 15–30.

Kutch, JJ and Valero-Cuevas, FJ (2012). "Challenges and new approaches to proving the existence of muscle synergies of neural origin". In: *PLoS computational biology* 8.5, e1002434.

Lee, DD and Seung, HS (2001). "Algorithms for non-negative matrix factorization". In: *Advances in neural information processing systems*, pp. 556–562.

Lemon, RN and Griffiths, J (2005). "Comparing the function of the corticospinal system in different species: organizational differences for motor specialization?" In: *Muscle & Nerve: Official Journal of the American Association of Electrodiagnostic Medicine* 32.3, pp. 261–279.

Lloyd, DPC (1943). "Neuron patterns controlling transmission of ipsilateral hind limb reflexes in cat". In: *Journal of Neurophysiology* 6.4, pp. 293–315.

Lundberg, A (1979). "Multisensory control of spinal reflex pathways". In: *Progress in brain research* 50, pp. 11–28.

Lundberg, A and Malmgren, K (1988). "The dynamic sensitivity of Ib inhibition". In: *Acta Physiol. Scand* 133, pp. 123–124.

Macefield, G, Hagbarth, KE, Gorman, R, Gandevia, SC, and Burke, D (1991). "Decline in spindle support to alpha-motoneurones during sustained voluntary contractions." In: *The Journal of physiology* 440.1, pp. 497–512.

Maier, MA, Illert, M, Kirkwood, PA, Nielsen, J, and Lemon, RN (1998). "Does a C3-C4 propriospinal system transmit corticospinal excitation in the primate? An investigation in the macaque monkey". In: *The Journal of physiology* 511.1, pp. 191–212.

Matthews, BHC (1933). "Nerve endings in mammalian muscle". In: *The Journal of Physiology* 78.1, pp. 1–53.

Matthews, PBC (1969). "Evidence that the secondary as well as the primary endings of the muscle spindles may be responsible for the tonic stretch reflex of the decerebrate cat". In: *The Journal of physiology* 204.2, pp. 365–393.

McCrea, DA (1992). "Can sense be made of spinal interneuron circuits?" In: *Behavioral and Brain Sciences* 15, pp. 633–633.

McCrea, DA and Rybak, IA (2008). "Organization of mammalian locomotor rhythm and pattern generation". In: *Brain research reviews* 57.1, pp. 134–146.

Milosevic, M, Yokoyama, H, Grangeon, M, Masani, K, Popovic, MR, Nakazawa, K, and Gagnon, DH (2017). "Muscle synergies reveal impaired trunk muscle coordination strategies in individuals with thoracic spinal cord injury". In: *Journal of Electromyography and Kinesiology* 36, pp. 40–48.

Mongeon, D, Blanchet, P, and Messier, J (2009). "Impact of Parkinson's disease and dopaminergic medication on proprioceptive processing". In: *Neuroscience* 158.2, pp. 426–440.

Nichols, TR, Bunderson, NE, and Lyle, MA (2016). "Neural regulation of limb mechanics: insights from the organization of proprioceptive circuits". In: *Neuromechanical modeling of posture and locomotion*. Springer, pp. 69–102.

Onishi, H, Yagi, R, Oyama, M, Akasaka, K, Ihashi, K, and Handa, Y (2002). "EMG-angle relationship of the hamstring muscles during maximum knee flexion". In: *Journal of Electromyography and Kinesiology* 12.5, pp. 399–406.

Osborne, H, Lai, YM, Lepperød, ME, Sichau, D, Deutz, L, and de Kamps, Marc (2021). "MIIND: A Model-Agnostic Simulator of Neural Populations". In: *Frontiers in Neuroinformatics* 15.

Pierrot-Deseilligny, Emmanuel and Burke, David (2005). *The circuitry of the human spinal cord: its role in motor control and movement disorders*. Cambridge university press.

Pincivero, DM, Salfetnikov, Y, Campy, RM, and Coelho, AJ (2004). "Angle-and gender-specific quadriceps femoris muscle recruitment and knee extensor torque". In: *Journal of biomechanics* 37.11, pp. 1689–1697.

Pratt, CA and Jordan, LM (1987). “Ia inhibitory interneurons and Renshaw cells as contributors to the spinal mechanisms of fictive locomotion”. In: *Journal of Neurophysiology* 57.1, pp. 56–71.

Pruszynski, JA and Scott, SH (2012). “Optimal feedback control and the long-latency stretch response”. In: *Experimental brain research* 218.3, pp. 341–359.

Rack, PMH and Westbury, DR (1969). “The effects of length and stimulus rate on tension in the isometric cat soleus muscle”. In: *The Journal of physiology* 204.2, pp. 443–460.

Rainoldi, A, Melchiorri, G, and Caruso, I (2004). “A method for positioning electrodes during surface EMG recordings in lower limb muscles”. In: *Journal of neuroscience methods* 134.1, pp. 37–43.

Rimini, D, Agostini, V, and Knaflitz, M (2017). “Intra-subject consistency during locomotion: similarity in shared and subject-specific muscle synergies”. In: *Frontiers in human neuroscience* 11, p. 586.

Roh, J, Rymer, WZ, and Beer, RF (2012). “Robustness of muscle synergies underlying three-dimensional force generation at the hand in healthy humans”. In: *Journal of neurophysiology* 107.8, pp. 2123–2142.

Rothwell, JC, Traub, MM, Day, BL, Obeso, JA, Thomas, PK, and Marsden, CD (1982). “Manual motor performance in a deafferented man”. In: *Brain* 105.3, pp. 515–542.

Safavynia, SA and Ting, LH (2013). “Long-latency muscle activity reflects continuous, delayed sensorimotor feedback of task-level and not joint-level error”. In: *Journal of neurophysiology* 110.6, pp. 1278–1290.

Saito, A and Akima, H (2015). “Neuromuscular activation of the vastus intermedius muscle during isometric hip flexion”. In: *Plos one* 10.10, e0141146.

Salenius, S, Portin, K, Kajola, M, Salmelin, R, and Hari, R (1997). “Cortical control of human motoneuron firing during isometric contraction”. In: *Journal of neurophysiology* 77.6, pp. 3401–3405.

Saltiel, P, Wyler-Duda, K, D’Avella, A, Tresch, MC, and Bizzi, E (2001). “Muscle synergies encoded within the spinal cord: evidence from focal intraspinal NMDA iontophoresis in the frog”. In: *Journal of neurophysiology* 85.2, pp. 605–619.

Santuz, A, Akay, T, Mayer, WP, Wells, TL, Schroll, A, and Arampatzis, A (2019). "Modular organization of murine locomotor pattern in the presence and absence of sensory feedback from muscle spindles". In: *The Journal of physiology* 597.12, pp. 3147–3165.

Sherrington, CS (1909). "Reciprocal innervation of antagonistic muscles. Fourteenth note.-On double reciprocal innervation". In: *Proceedings of the Royal Society of London. Series B, Containing Papers of a Biological Character* 81.548, pp. 249–268.

Shevtsova, NA, Hamade, K, Chakrabarty, S, Markin, SN, Prilutsky, BI, and Rybak, IA (2016). "Modeling the organization of spinal cord neural circuits controlling two-joint muscles". In: *Neuromechanical Modeling of Posture and Locomotion*. Springer, pp. 121–162.

Shinoda, Y, Yokota, J, and Futami, T (1981). "Divergent projection of individual corticospinal axons to motoneurons of multiple muscles in the monkey". In: *Neuroscience letters* 23.1, pp. 7–12.

Thomas, HO (1878). *Diseases of the hip, knee, and ankle joints: with their deformities, treated by a new and efficient method*. HK Lewis.

Torres-Oviedo, G, Macpherson, JM, and Ting, LH (2006). "Muscle synergy organization is robust across a variety of postural perturbations". In: *Journal of neurophysiology* 96.3, pp. 1530–1546.

Torres-Oviedo, G and Ting, LH (2007). "Muscle synergies characterizing human postural responses". In: *Journal of neurophysiology* 98.4, pp. 2144–2156.

Tresch, MC and Bizzi, E (1999). "Responses to spinal microstimulation in the chronically spinalized rat and their relationship to spinal systems activated by low threshold cutaneous stimulation". In: *Experimental brain research* 129.3, pp. 401–416.

Tresch, MC, Cheung, VCK, and d'Avella, A (2006). "Matrix factorization algorithms for the identification of muscle synergies: evaluation on simulated and experimental data sets". In: *Journal of neurophysiology* 95.4, pp. 2199–2212.

Tresch, MC, Saltiel, P, d'Avella, A, and Bizzi, E (2002). "Coordination and localization in spinal motor systems". In: *Brain Research Reviews* 40.1-3, pp. 66–79.

Tresch, MC, Saltiel, P, and E, Bizzi (1999). "The construction of movement by the spinal cord". In: *Nature neuroscience* 2.2, pp. 162–167.

Wilson, HR and Cowan, JD (1972). "Excitatory and inhibitory interactions in localized populations of model neurons". In: *Biophysical journal* 12.1, pp. 1–24.

Yang, Q, Logan, D, and Giszter, SF (2019). "Motor primitives are determined in early development and are then robustly conserved into adulthood". In: *Proceedings of the National Academy of Sciences*, p. 201821455.

Zabik, RM and Dawson, ML (1996). "Comparison of force and peak EMG during a maximal voluntary isometric contraction at selected angles in the range of motion for knee extension". In: *Perceptual and motor skills* 83.3, pp. 976–978.

Chapter 5

Discussion and Conclusions

In the first two articles presented here, we have seen the development of a compelling new simulation technique. The MIIND grid method allows modellers to generate population-level behaviour from their own custom high-dimensional neural models. Simulations remain efficient to run, even on an individual PC, and require only the definition of the underlying neuron model and an intuitive description of the population network. In contrast to other PDTs, the full density distribution across all time-dependent variables is calculated enabling a full understanding of the population dynamics. From a technical perspective, MIIND can now be installed quickly and easily, integrated with Python, and used to produce pertinent population-level results and engaging visualisations of the population distribution. The third article demonstrated a MIIND simulation of neural populations in the spinal cord to support results from a novel isometric extension task experiment. Analysis of both the experimental and simulated results indicates that afferent signals significantly influence activation across a range of muscles during a more realistic isometric task than traditional experiments. The results show that it may be important to take proprioception into account when developing therapies and function tests for those with motor impairment. Though each article covers the majority of pertinent discussion points, additional details and some further ideas that have occurred since publication are raised here.

5.1 Visualising the Mesh and Grid Methods

Papers A and B, while giving the technical implementation details of the mesh and grid methods, do not give an account of the mathematical underpinning of the technique as this is described thoroughly in other articles (de Kamps, 2003; de Kamps, 2013). However, the algorithm for solving the time evolution of the probability density function in code is surprisingly simple compared to what might be considered a brute-force approach based on the formulae. The important steps in the derivation of this technique are the use of the method of characteristics to transform and simplify the partial differential equation describing the evolution of the probability density function that appears in earlier PDTs, and a second transformation allowing the complete separation of the deterministic and non-deterministic dynamics. Appendix sections A.1, A.2, and A.3 show how these steps are achieved and describe the Chapman-Kolmogorov equation which leads to the master equation definition of the Poisson noise process.

The probability density function, $\rho(\vec{v}, t)$, over a neuron state, $\vec{v}$, at time, t, in the mesh and grid methods evolves in time according to equation 5.2 when values of $\vec{v}$ are chosen according to a characteristic curve of the underlying neuron model, $\vec{F}$. The chosen values are defined by the function $\vec{v}'(t)$ of equation 5.1. In other words, $\vec{v}'$ represents a coordinate transformation.

$$\frac{d\vec{v}'}{dt} = \frac{\vec{F}}{\tau} \tag{5.1}$$

$$\frac{d\rho}{dt} = -\frac{\rho}{\tau}\frac{\partial \vec{F}}{\partial \vec{v}'} \tag{5.2}$$

A further transformation of ρ to ρ' sets the time evolution to zero.

$$\rho' = e^{\int \frac{1}{\tau}\frac{\partial \vec{F}}{\partial \vec{v}} dt} \rho \tag{5.3}$$

What do the transformations in equations 5.1 and 5.3 do to a given ρ? Fig. 5.1A shows an arbitrary probability density function constructed from the sum of two Gaussians at a time, $t = 0$. For this example, we consider the state to be a single value, v, representing the membrane potential of a leaky neuron with $F = -v$. The integration of incoming spikes and spiking behaviour are considered separately. By solving the two ordinary differential equations 5.1 and 5.2, ρ correctly moves in time and approaches a Dirac delta peak at $v = 0$. The discretisation line shows a characteristic curve starting somewhat arbitrarily at $v = v_0 = 2.0$. The discretisation line is the solution to the definition of v'. Fig. 5.1B shows the result of transforming ρ into v'-space. The discretisation line has been similarly transformed resulting in a set of equally spaced points. Notice that, in both the transformed and non-transformed plots, with each increase in time, ρ is shifted an equal number of points along the characteristic curve discretisation. However, only in the transformed space does this appear as a constant shift to the right. This is a desirable simplification for solving the system. In v'-space, the change in ρ becomes that defined in equation 5.2. As the distribution moves to the right in the transformed space and approaches zero in the non-transformed space, the height of the distribution increases to maintain the probability density in the non-transformed space as its width decreases (which can be seen with increasingly small discretisation bins). The transformation to ρ' eliminates this change. Therefore in the transformed space, the distribution remains constant in time except for the shift to the right. In the non-transformed space, the height remains the same, indicating that ρ' is not a probability density function but a probability mass function.

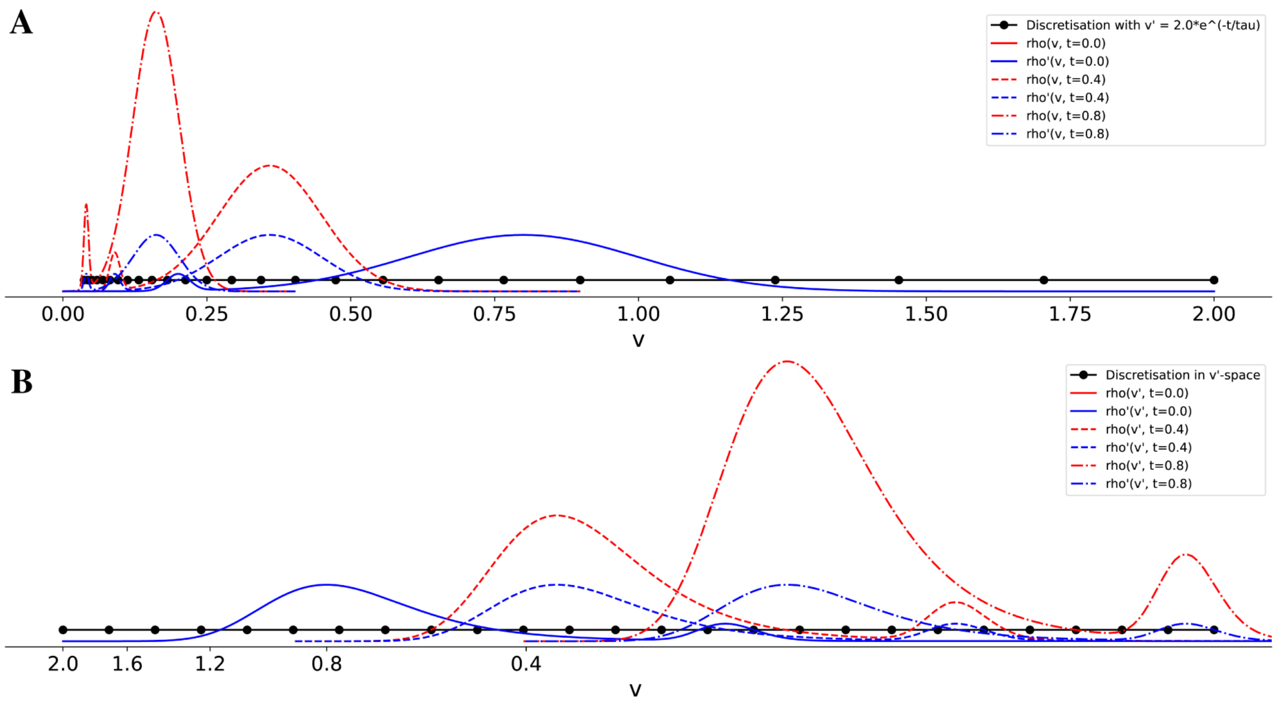


Figure 5.1: (A) The probability density function, ρ, and the transformed ρ' for $t = 0$, $t = 0.4$, and $t = 0.8$ in the non-transformed v-space. $\rho(v, t = 0)$ is an arbitrary function for demonstration purposes build from the sum of two Gaussians ($\mu_1 = 0.8, \sigma_1 = 0.2, \mu_2 = 0.2, \sigma_2 = 0.02$). The neuron function is defined as, $F = -v$ with $\tau = 0.5$. The discretisation curve shows the solution to the transformation of v defined in equation 5.1 with $v_0 = 2$. (B) Along the characteristic curve under the coordinate transformation, $\vec{v}'$, the discretisation becomes a set of regularly separated points. At the three different time points, ρ and ρ' are shifted to the right by a constant amount.

In code, the two transformations can be performed in an elegant and implicit manner. As with many numerical methods, the function ρ must be discretised so it can be represented in memory. In the mesh method, the characteristic curves are used to define a discretisation of ρ. The arbitrary choice for v_0 in the solution to equation 5.1 should be chosen so that the curve covers all values of v which are expected to have a non-zero probability density during a simulation. The discretisation can be generated by using a time-stepping algorithm to integrate the ODE of equation 5.1. Crucially, by associating each value of ρ with the differently sized volumes of the discretisation in an array, the transformation from $\vec{v}$ to $\vec{v}'$ is made implicitly. The regularity of the transformed discretisation in Fig. 5.1B means that it can be considered as a set of addresses in memory. However, every value in the array must be updated to the next position along in memory after each time interval to correctly implement the solution to the change in v. As discussed in paper A, memory updates such as this can be extremely fast when using pointers in C++. The time evolution of ρ in equation 5.2 maintains the correct probability which is shown in Fig. 5.1. However, the second transformation specifically inverts this change so that the noise process can be applied. In the value array of ρ, this is equivalent to performing no additional calculation on the values. Therefore, without further code execution, the values in the array already represent ρ'. The second transformation is a no-op. If at any point, the density function ρ is required, each value of ρ' can be multiplied by the associated bin width in the non-transformed discretisation.

5.1.1 Solving for ρ with the Grid Method

The discretisation of the state space in the grid method is uniform, therefore, the implicit coordinate transformation of equation 5.1 is not performed. Instead, the movement of probability mass due to the dynamics, $\frac{\vec{F}}{\tau}$, is performed by the transition matrix. As with the mesh method, the change to the probability density is not performed. Therefore, the probability mass function calculated by the grid method is ρ' but it remains in $\vec{v}$-space. It is clear from Fig. 5.1 that equation 5.4 holds under the two transformations.

$$\rho'(\vec{v}, t + \delta t) = \rho'(\vec{v} + \delta t \frac{F(\vec{v})}{\tau}, t) \tag{5.4}$$

It remains true in the limit as $\delta t \to 0$.

$$\frac{d\rho'}{dt} = \lim_{\delta t \to 0} \frac{\rho'(\vec{v} + \delta t \frac{F(\vec{v})}{\tau}, t) - \rho'(\vec{v}', t)}{\delta t} \tag{5.5}$$

Because $\delta t \frac{F(\vec{v})}{\tau}$ is time-invariant, the Chapman-Kolmogorov equation described in the Appendix section A.3 holds and the solution simplifies to the master equation A.14 but with $\mathbf{Q}$ equal to a matrix which stores the transition probabilities between bins of a regular (grid) discretisation after a single time step of the underlying neuron model, $\frac{\vec{F}}{\tau}$. Therefore, in the grid method, the process of solving the master equation can be applied to both the deterministic and non-deterministic parts. The resulting discrete representation of equation 5.5 is given by equation 5.6. P_k is the probability mass of a given cell, k, in the grid. α_{lk} is the proportion of overlap between cell k and each other grid cell, l, after it has been transformed according to $dt\frac{\vec{F}}{\tau}$.

$$\frac{dP_k}{dt} = \sum_{l \neq k} (\alpha_{lk} P_l) - P_k \tag{5.6}$$

5.2 Possible Simplifications or Extensions for the Grid Method

Since both elements of the grid method (the deterministic dynamics and Poisson process) can be represented as a master equation. If the transition matrix is held constant, both equations can be combined as in equation 5.7 and solved at once.

$$\frac{d\mathbf{P}(t)}{dt} = \lambda \mathbf{P}(t)\mathbf{R}\mathbf{Q} \tag{5.7}$$

Where $\mathbf{R}$ is the transition matrix of the noise process and $\mathbf{Q}$ is the transition matrix of the deterministic dynamics. As both of these matrices remain constant throughout the simulation, the matrix $\mathbf{RQ}$ need only be calculated once. This simplification would certainly improve computational performance of the technique. However, as discussed, the overriding limitation on efficiency is the number of cells in the grid which grows exponentially with dimensionality. It also relies on the time step, dt being small enough that solving with the Euler method remains stable.

Instead of simplifying, one can recall that in the mesh method, the reason that the master equation is required to solve the distribution of probability mass due to Poisson input is because of the discretisation of the state space along the characteristic curves. The spread of mass between cells according to the Poisson distribution is not uniform. In other words, the master equation gives the effect of the non-deterministic noise process on the transformed probability density function, ρ', of equation 5.3. An alternative method for solving the Poisson process available in the grid method, however, is to solve the master equation for just a single cell with probability mass equal to 1 and then perform a convolution on the full grid. This alternative method is generally not more efficient as the number of updates to the mass function (the size of the distribution window multiplied by the number of cells in the grid) is similar to the number required to multiply the transition matrix by the mass vector when solving the master equation. Nevertheless, any distribution that can be precalculated and convolved with the grid in this manner can be modelled in the grid method potentially enabling other noise distributions.

5.3 More Ideas for Addressing Memory Usage and the Curse of Dimensionality

Both the mesh and grid implementations in MIIND are an exercise in trading computation time for memory usage. The deterministic dynamics of the underlying neuron model, whether stored in memory as a mesh or as a transition matrix, have been pre-solved so that during simulation a small number of memory lookups take the place of calculating the movement of probability mass. In two dimensions, the space taken in memory of a very detailed mesh or high-resolution grid remains manageable. Generally, the time to generate the pre-calculated files and load them into memory at the beginning of each simulation is dominated by the time required to perform the updates per cell per time step. As noted in papers A and B, the memory requirements for the two-dimensional mesh and transition matrices are an order of magnitude lower than those required to hold the spike history in direct simulation techniques. However, for higher grid dimensions, loading the transition matrix into memory at the start of a simulation can take longer than the simulation itself. One remedy might be to generate and store a lower resolution transition matrix and, during simulation time, subdivide each large grid cell into smaller ones using a linear interpolation function on the assumption that the neural dynamics only change smoothly between large cells. This will not fully solve the problem, however, and even if the memory limits are not reached, the time required to update every cell in the grid will continue to rise exponentially with each additional dimension. As discussed in paper B, other PDTs have been developed to avoid this problem by tracking the age of the neuron rather than its membrane potential or other state variables. This drastically limits the flexibility of the technique but it is clear that dimensionality reduction must be employed in some fashion to improve the efficiency of the grid and mesh methods. Artificial neural networks (ANNs) have been used for many years to perform such reductions. Auto-encoders (Cottrell and Munro, 1988) are ANNs with a reduced number of hidden neurons (or nodes) which are trained to reproduce their input. If successful, the small number of hidden nodes have encoded the much larger number of input nodes resulting in a dimensional reduction. Training an auto-encoder on a transition matrix might yield a reduction so that only the values of the hidden nodes need be stored along with the second half of the ANN to rebuild the matrix during simulation time. Obviously, requiring the transition matrix for training and the need to decode at simulation time means that this process would not avoid long pre-processing times and might degrade simulation speed. Nonetheless, it would alleviate some of the pressure on memory from an increased number of dimensions. However, there are a great many other techniques involving ANNs to be investigated. For example, a multi-layer ANN could be trained on the individual transitions or perhaps even the state changes of individual neurons which might yield a smaller memory footprint than the full transition matrix. In any future reduction technique, we want to be able to maintain or recover the full distribution across state space so that the link between the underlying model and the population is not lost. Recent work into normalising flows (Tabak and Vanden-Eijnden, 2010; Rezende and Mohamed, 2015) points to techniques for deep learners to find a low dimensional function for mapping simple distributions to distributions with more complex or even unknown definitions. Once learned, such functions could be applied in MIIND in place of applying the transition matrix which could represent a true dimensional reduction and significantly improve performance.

5.4 Better Applications of the Grid Method to Spinal Cord Models

With the definition of a probability density function, rather than individual neurons, the mesh and grid methods make the implicit assumption that there are an infinite number of neurons in a population. This is appropriate for simulations of cortical patches or large neural structures in the brain as variation in the population activity is likely to be minimal with many tens of thousands of neurons. However, the spinal cord model of paper C and the Potjans-Diesmann model presented in paper A both represent populations of fewer than 10000 neurons. There is an increased likelihood of large fluctuations in activity-dependent on the specific realisation of input spike trains and synchronous firing which cannot be captured by the PDT. In paper C, the results from NMF applied to the firing rate activity of the motor neuron populations in the model were compared to those applied to the sEMG recordings of the muscles themselves. Crucial to the conclusions of the study is the assumption that these results are comparable. However, there is little evidence that sEMG signals can be used to infer the firing rate of the alpha motor neurons controlling a given muscle. Instead, the amplitude of the raw sEMG signal can be interpreted, particularly in the larger muscles, as representative of the proportion of active motor units (Negro and Farina, 2012; Dideriksen et al., 2018). Lower motor neurons can be classified according to their size (by soma area and axon width). It has been found that with increasing size, motor neuron excitability and resistance to fatigue reduces and the number of affected muscle fibres increases leading to greater force production (Milner-Brown et al., 1973; Burke et al., 1973; Kanning et al., 2010). As a result, there is an ordering of motor units recruited for a task as the force requirement increases. For a maximal voluntary contraction (MVC) task, it is expected that all motor neurons are recruited in size order. In the vicinity of a surface electrode, a greater number of active motor units (accounting for constructive and destructive interference) should produce a greater recorded amplitude. There is not necessarily a correlation between sEMG amplitude and the firing rate of motor neurons. There is correlation with the input firing rates to the motor neurons because this is what affects the number of active motor units due to that range of excitability. In the spinal circuit model of paper C, this relationship is not accounted for. However, because the motor neuron populations are based on simple integrate-and-fire neurons, each population firing rate represents the aggregation of all inputs and so a comparison of the results of the two experiments can be made. One might argue that a rate-based population model may have been better for this study without the use of a PDT. However, at the beginning of the study, there was no way to know if certain transient effects (such as can be seen in Fig. 3.9 of paper B) might be visible in the sEMG data which a rate-based approach would not have produced. In future studies, and with the benefit of the higher-dimensional grid PDT, work should be done to develop a population model that explicitly captures motor unit recruitment dependent on firing rate input and excitability. This would be a departure from the traditional use of MIIND where each variable refers to the state of the individual neuron with the assumption that the population is homogeneous. An added excitability dimension in the grid would render the population heterogeneous in that parameter. Recording the average membrane potential of this population might prove a closer and more biologically plausible approximation to sEMG activity.

5.5 Improving Mechanical Analysis of Motor Control Experiments

As mentioned, the experimental protocol of the isometric task performed in paper C was newly designed, based on existing clinical tests. Careful consideration was required to identify that the rectus femoris muscle if truly maximally activated, would have lifted the leg off the table due to its biarticular nature (it extends the knee and flexes the hip). Furthermore, in developing the model, working out which afferent signals should increase and which should decrease with a change in knee angle was based on an estimate of how the length of each muscle should change. Had the neural model been integrated with a musculoskeletal model such as those available in OpenSim (Delp et al., 2007) or the Neuro-Robotics Platform (Falotico et al., 2017), these considerations would have been immediately obvious. In combination with MIIND, both platforms have the potential to produce "closed-loop" limb movement simulations with Hill-type muscles (Millard et al., 2013) and mathematical models of muscle spindles and Golgi tendon organs for proprioceptive feedback (Mileusnic, Brown, et al., 2006a; Mileusnic and Loeb, 2006b). Such a system could prove invaluable for motor control experiment design and analysis.

5.6 Using Experimental Data in MIIND

All three articles demonstrate the use of MIIND to simulate populations of point model neurons. Similar to other PDTs, development of the technique has focused on common neuron models such as leaky integrate-and-fire or adaptive exponential integrate-and-fire. Recently, however, other PDTs (Schwalger, Deger, et al., 2017) have been able to simulate populations of so-called generalised integrate-and-fire (GIF) neurons (Mihalaş and Niebur, 2009; Teeter et al., 2018). GIF neurons are set up similarly to LIF neurons but with a movable spike threshold which is dependent on incoming currents and a series of update rules. They are particularly useful because the parameters can be set to match many phenomenological behaviours of real neurons with a reduced set of time-dependent variables. No major changes would be required to implement populations of such neurons in MIIND except to replace the currently constant threshold/reset mapping with a time and voltage-dependent one. This would be possible in both the grid and mesh methods although it would be less detrimental to the speed of the grid method. However, the number of time-dependent input currents required for more complex behaviours (for so-called type 3 to type 5 GIF neurons) would result in up to five dimensions in total.

5.7 Closing Remarks

As technology improves, the speed, size, and complexity of Monte Carlo neural simulations increases. Development of refractory density methods is enabling finite-size effects and low dimensional firing rate dynamics. Thanks to the grid method, the MIIND PDT, though based on an older technique, holds its own in terms of efficiency and flexibility. There remains significant scope for further development while retaining the strongest of links to the underlying neural dynamics.

Applying PDTs to spinal cord circuit models is a completely new approach which encourages the development of simple but powerful neural population models over the complexities of direct simulation. The flexibility of the grid method means that we can start to consider heterogeneous population models by combining the variables of the underlying neurons with different parameter ranges to better approximate experimental observations.

Finally, the ability to easily render the density distribution in a MIIND simulation gives a unique insight into the behaviour of a population in contrast to that of individual neurons. It could prove a valuable tool for teaching others about dynamics and noise in neuroscience and many other fields.

References

Apfaltrer, Felix, Ly, Cheng, and Tranchina, Daniel (2006). "Population density methods for stochastic neurons with realistic synaptic kinetics: Firing rate dynamics and fast computational methods". In: *Network: Computation in Neural Systems* 17.4, pp. 373–418.

Bolognini, Nadia, Russo, Cristina, and Edwards, Dylan J (2016). "The sensory side of post-stroke motor rehabilitation". In: *Restorative neurology and neuroscience* 34.4, pp. 571–586.

Burke, RE, Levine, DN, Tsairis, P, and Zajac Iii, FE (1973). "Physiological types and histochemical profiles in motor units of the cat gastrocnemius". In: *The Journal of physiology* 234.3, pp. 723–748.

Chizhov, Anton V, Graham, Lyle J, and Turbin, Andrey A (2006). "Simulation of neural population dynamics with a refractory density approach and a conductance-based threshold neuron model". In: *Neurocomputing* 70.1-3, pp. 252–262.

Cottrell, Garrison W and Munro, Paul (1988). "Principal components analysis of images via back propagation". In: *Visual Communications and Image Processing'88: Third in a Series*. Vol. 1001. SPIE, pp. 1070–1077.

Crone, C, Nielsen, J, Petersen, N, Ballegaard, M, and Hultborn, H (1994). "Disynaptic reciprocal inhibition of ankle extensors in spastic patients". In: *Brain* 117.5, pp. 1161–1168.

de Kamps, Marc (2003). "A simple and stable numerical solution for the population density equation". In: *Neural computation* 15.9, pp. 2129–2146.

– (2013). "A generic approach to solving jump diffusion equations with applications to neural populations". In: *arXiv preprint arXiv:1309.1654*. URL: `https://arxiv.org/abs/1309.1654v2`.

de Kamps, Marc, Lepperød, Mikkel, and Lai, Yi Ming (2019). “Computational geometry for modeling neural populations: From visualization to simulation”. In: *PLoS computational biology* 15.3, e1006729.

Delp, Scott L, Anderson, Frank C, Arnold, Allison S, Loan, Peter, Habib, Ayman, John, Chand T, Guendelman, Eran, and Thelen, Darryl G (2007). “OpenSim: open-source software to create and analyze dynamic simulations of movement”. In: *IEEE transactions on biomedical engineering* 54.11, pp. 1940–1950.

Dideriksen, Jakob L, Negro, Francesco, Falla, Deborah, Kristensen, Signe R, Mrachacz-Kersting, Natalie, and Farina, Dario (2018). “Coherence of the surface EMG and common synaptic input to motor neurons”. In: *Frontiers in Human Neuroscience* 12, p. 207.

Falotico, Egidio, Vannucci, Lorenzo, Ambrosano, Alessandro, Albanese, Ugo, Ulbrich, Stefan, Vasquez Tieck, Juan Camilo, Hinkel, Georg, Kaiser, Jacques, Peric, Igor, Denninger, Oliver, et al. (2017). “Connecting artificial brains to robots in a comprehensive simulation framework: the neurorobotics platform”. In: *Frontiers in neurorobotics* 11, p. 2.

Gewaltig, Marc-Oliver and Diesmann, Markus (2007). “NEST (NEural Simulation Tool)”. In: *Scholarpedia* 2.4, p. 1430.

Hultborn, H, Meunier, S, Pierrot-Deseilligny, E, and Shindo, M (1987). “Changes in presynaptic inhibition of Ia fibres at the onset of voluntary contraction in man.” In: *The Journal of physiology* 389.1, pp. 757–772.

Iyer, Ramakrishnan, Menon, Vilas, Buice, Michael, Koch, Christof, and Mihalas, Stefan (2013). “The influence of synaptic weight distribution on neuronal population dynamics”. In: *PLoS computational biology* 9.10, e1003248.

Johannesma, Petrus Ignatius Marie (1969). “Stochastic neural activity: A theoretical investigation”. PhD thesis. Nijmegen: Faculteit der Wiskunde en Natuurwetenschappen.

Kanning, Kevin C, Kaplan, Artem, and Henderson, Christopher E (2010). “Motor neuron diversity in development and disease”. In: *Annual review of neuroscience* 33, pp. 409–440.

Knight, Bruce W (1972). “Dynamics of encoding in a population of neurons”. In: *The Journal of general physiology* 59.6, pp. 734–766.

Knight, James C, Komissarov, Anton, and Nowotny, Thomas (2021). “PyGeNN: A Python Library for GPU-Enhanced Neural Networks”. In: *Frontiers in Neuroinformatics* 15.

Matthews, PBC (1969). “Evidence that the secondary as well as the primary endings of the muscle spindles may be responsible for the tonic stretch reflex of the decerebrate cat”. In: *The Journal of physiology* 204.2, pp. 365–393.

Mihalaş, Ştefan and Niebur, Ernst (2009). “A generalized linear integrate-and-fire neural model produces diverse spiking behaviors”. In: *Neural computation* 21.3, pp. 704–718.

Mileusnic, Milana P, Brown, Ian E, Lan, Ning, and Loeb, Gerald E (2006a). “Mathematical models of proprioceptors. I. Control and transduction in the muscle spindle”. In: *Journal of neurophysiology* 96.4, pp. 1772–1788.

Mileusnic, Milana P and Loeb, Gerald E (2006b). “Mathematical models of proprioceptors. II. Structure and function of the Golgi tendon organ”. In: *Journal of neurophysiology* 96.4, pp. 1789–1802.

Millard, Matthew, Uchida, Thomas, Seth, Ajay, and Delp, Scott L (2013). “Flexing computational muscle: modeling and simulation of musculotendon dynamics”. In: *Journal of biomechanical engineering* 135.2.

Milner-Brown, HS, Stein, RB, and Yemm, R_ (1973). “The orderly recruitment of human motor units during voluntary isometric contractions”. In: *The Journal of physiology* 230.2, p. 359.

Negro, Francesco and Farina, Dario (2012). “Factors influencing the estimates of correlation between motor unit activities in humans”. In.

Omurtag, Ahmet, Knight, Bruce W, and Sirovich, Lawrence (2000). “On the simulation of large populations of neurons”. In: *Journal of computational neuroscience* 8.1, pp. 51–63.

Rezende, Danilo and Mohamed, Shakir (2015). “Variational inference with normalizing flows”. In: *International conference on machine learning*. PMLR, pp. 1530–1538.

Schuhfried, Othmar, Crevenna, Richard, Fialka-Moser, Veronika, and Paternostro-Sluga, Tatjana (2012). “Non-invasive neuromuscular electrical stimulation in patients with central nervous system lesions: an educational review”. In: *Journal of rehabilitation medicine* 44.2, pp. 99–105.

Schwalger, Tilo, Deger, Moritz, and Gerstner, Wulfram (2017). “Towards a theory of cortical columns: From spiking neurons to interacting neural populations of finite size”. In: *PLoS computational biology* 13.4, e1005507.

Schwalger, Tilo and Lindner, Benjamin (2015). “Analytical approach to an integrate-and-fire model with spike-triggered adaptation”. In: *Physical Review E* 92.6, p. 062703.

Tabak, Esteban G and Vanden-Eijnden, Eric (2010). “Density estimation by dual ascent of the log-likelihood”. In: *Communications in Mathematical Sciences* 8.1, pp. 217–233.

Teeter, Corinne, Iyer, Ramakrishnan, Menon, Vilas, Gouwens, Nathan, Feng, David, Berg, Jim, Szafer, Aaron, Cain, Nicholas, Zeng, Hongkui, Hawrylycz, Michael, et al. (2018). “Generalized leaky integrate-and-fire models classify multiple neuron types”. In: *Nature communications* 9.1, pp. 1–15.

Appendix A

MIIND's Mathematical Derivation

A.1 The PDT advection formula

It is instructive to see how the software algorithm implemented in MIIND relates to the mathematical underpinning. de Kamps, 2013 provides the full derivation of the technique and we begin in the same place by considering a general definition for the changing state of a neuron in time.

$$\tau \frac{d\vec{v}}{dt} = \vec{F}(\vec{v}) \tag{A.1}$$

The vector $\vec{v}$ describes the state of the neuron. This can be as simple as a single value representing the membrane potential as in the leaky integrate-and-fire model, or a larger set such as the gating variables of the Hodgkin-Huxley neuron model. The function, $\vec{F}$, describes how the state changes with time on a time scale, τ. If we consider a homogeneous population of such neurons, a probability density function, $\rho(\vec{v}, t)$, can be defined which describes the likelihood of finding a neuron from the population with a given state, $\vec{v}$, at a given time, t. It provides the probability of finding a neuron from the population with a state within a range of states, $d\vec{v}$. We can further define an updated function, $\rho(\vec{v}', t')$, where $\vec{v}' = \vec{v} + d\vec{v}$ and $t' = t + dt$. Equation A.2 gives a general solution to $\rho(\vec{v}', t')$ in terms of $\rho(\vec{v}, t)$.

$$\rho(\vec{v}', t') = \rho(\vec{v}, t) + dt \frac{\partial \rho(\vec{v}, t)}{\partial t} + d\vec{v}. \frac{\partial \rho(\vec{v}, t)}{\partial \vec{v}} \tag{A.2}$$

From this definition, we would like to find an approximation to the time derivative of $\rho(\vec{v}, t)$. The conservation of probability requires the following.

$$\rho(\vec{v}', t')d\vec{v}' = \rho(\vec{v}, t)d\vec{v} \tag{A.3}$$

To eliminate $d\vec{v}'$, we presume some matrix, J, which transforms $d\vec{v}$ to $d\vec{v}'$. We can use the Taylor expansion approximated to the first two terms.

$$\begin{aligned} d\vec{v}' &= |J|d\vec{v} \\ |J| &= |\frac{d\vec{v}'}{d\vec{v}}| \\ |J| &= |\mathbb{I} + \frac{1}{\tau}\frac{\partial \vec{F}(\vec{v})}{\partial \vec{v}}dt| \end{aligned} \tag{A.4}$$

Substituting into equation A.3 gives

$$\rho(\vec{v}, t)d\vec{v} = \rho(\vec{v}', t')|\mathbb{I} + \frac{1}{\tau}\frac{\partial \vec{F}(\vec{v})}{\partial \vec{v}}dt|d\vec{v} \tag{A.5}$$

Substituting equation A.2 then collecting terms up to $O(dt)$, gives an approximation for the time evolution of ρ.

$$\frac{\partial \rho}{\partial t} = -\frac{1}{\tau}(\frac{\partial \vec{F}}{\partial \vec{v}}\rho + \vec{F}\frac{\partial \rho}{\partial \vec{v}}) \tag{A.6}$$

$$\frac{\partial \rho}{\partial t} = -\frac{1}{\tau}\frac{\partial \vec{F}\rho}{\partial \vec{v}} \tag{A.7}$$

A.2 Derivation of the Mesh and Grid Methods

This advection formula describes the time evolution of the probability density function, ρ, over possible neuron states, $\vec{v}$, due to the underlying neural dynamics (the neuron model), $\vec{F}$. de Kamps, 2003 and Iyer et al., 2013 identified a transformation of equation A.6 which leads to an efficient algorithm for solving the time evolution of the neural dynamics. Firstly, the advection formula can be transformed into a set of ordinary differential equations using the method of characteristics. We first define the the derivative of $\rho(\vec{v}(s), t(s))$ with respect to a parameter, s. Equation A.8 gives the general definition which can be written in a form that matches the two terms involving $\partial\rho$ from equation A.7.

$$\frac{d\rho(\vec{v},t)}{ds} = \frac{\partial \rho}{\partial \vec{v}}\frac{d\vec{v}}{ds} + \frac{\partial \rho}{\partial t}\frac{dt}{ds} = -\frac{\rho}{\tau}\frac{\partial \vec{F}}{\partial \vec{v}} \tag{A.8}$$

By inspection, $\frac{dt}{ds} = 1$ and choosing $t(0) = 0$, we can set $s = t$. The resulting ordinary differential equations are as follows.

$$\frac{d\vec{v}}{dt} = \frac{\vec{F}}{\tau} \tag{A.9}$$

$$\frac{d\rho}{dt} = -\frac{\rho}{\tau}\frac{\partial \vec{F}}{\partial \vec{v}} \tag{A.10}$$

This set of equations tells us that along the characteristic curves given by $\frac{d\vec{v}}{dt}$, ρ changes according to $\frac{d\rho}{dt}$. ρ can be transformed to ρ' so that $\frac{d\rho'}{dt} = 0$ along the curves.

$$\rho' = e^{\int \frac{1}{\tau}\frac{\partial \vec{F}}{\partial \vec{v}} dt}\rho \tag{A.11}$$

With ρ' constant along the characteristic curves, a noise process can be applied to the transformed density function to represent the effect of random incoming spikes.

A.3 The Chapman-Kolmogorov equation and the Poisson master equation

After the transformation to ρ', if a noise term is then added to represent the effect of incoming random spikes on the population, the definition captures the full dynamics. We make the assumption that a single incoming spike to a neuron in the population causes an instantaneous change in state (usually an increase or decrease in membrane potential). The following Chapman-Kolmogorov equation describes the time evolution of ρ' due to incoming Poisson distributed spikes.

$$\frac{d\rho'}{dt} = \int_{M'} d\vec{w}' \left\{ W(\vec{v}'|\vec{w}')\rho'(\vec{w}') - W(\vec{w}'|\vec{v}')\rho'(\vec{v}') \right\} \tag{A.12}$$

$W(\vec{v}'|\vec{w}')$ gives the probability that a neuron with state $\vec{w}'$ has received a spike and now has a state

$\vec{v}'$. The right-hand side, therefore, calculates the change in ρ' due to neurons entering and leaving a volume $d\vec{w}'$. A derivation of the discrete version of this equation follows.

A homogeneous Markovian stochastic process describes how a random variable changes its value or state over discrete time steps. For each possible state, the random variable will enter a new state in the next time step with a given probability which can be represented as a Markov chain or as a probability distribution, $P(t)$ after a number of discrete time steps t. Alternatively, if an initial probability distribution, $P(0)$ is defined across states, $P(t)$ represents the probability distribution after t steps. The process is Markovian if the next state of the random variable is only dependent on the current state. It is homogeneous if the probability distribution remains constant for each time step. For such a case, the Chapman-Kolmogorov equation states that $P(t + dt) = P(t).P(dt)$. In other words, the transition matrix, which describes the probability distribution for transitioning from one state to the next, can be applied n times to find the probability of reaching each state n steps in the future. In the limit as $dt \to 0$, the following differential equation describes how the probability of being in each state changes with increasing time.

$$\frac{dP(t)}{dt} = \lim_{\delta t \to 0} \frac{P(t + \delta t) - P(t)}{\delta t} \tag{A.13}$$

Substituting the Chapman-Kolmogorov equation, the following differential equation immediately follows.

$$\frac{d\mathbf{P}(t)}{dt} = \mathbf{P}(t)\mathbf{Q} \tag{A.14}$$

This is a master equation with a jump-rate matrix $\mathbf{Q}$.

$$\mathbf{Q} = \lim_{\delta t \to 0} \left[\frac{\mathbf{P}(\delta t) - \mathbf{I}}{\delta t} \right]$$

For a monotonically accumulating count of Poisson events with rate parameter λ, $\mathbf{Q}$ looks like the following.

$$\mathbf{Q} = \begin{bmatrix} -\lambda & \lambda & 0 & 0 & \dots \\ 0 & -\lambda & \lambda & 0 & \dots \\ 0 & 0 & -\lambda & \lambda & \dots \\ \vdots & \vdots & \vdots & \vdots & \end{bmatrix} \tag{A.15}$$

If one event has occurred by time t, the probability of the count remaining at 1 reduces exponentially with parameter $-\lambda$ and the probability of the count increasing to 2 increases exponentially with parameter λ. Once 1 event has occurred, the state cannot move back to 0 events and it cannot skip to 3 events. Not including itself, each state has one state which leads to it and one state to which it leads. The outgoing transition probabilities of every state must sum to 1. Therefore, the rows of the matrix, $\mathbf{Q}$, each sum to zero.

$$\begin{aligned} \sum_{l}(Q_{kl}P_k) &= 0 \\ Q_{kk}P_k &= -\sum_{l\neq k}(Q_{kl}P_k) \end{aligned} \tag{A.16}$$

The master equation for a single state, k in P gives

$$\begin{aligned} \frac{dP_k}{dt} &= \sum_{l} Q_{lk}P_l \\ \frac{dP_k}{dt} &= \sum_{ll\neq k}(Q_{lk}P_l - Q_{kl}P_k) \end{aligned} \tag{A.17}$$

Substituting the result of equation A.16 gives

$$\frac{dP_k}{dt} = \sum_{ll\neq k}(Q_{lk}P_l - Q_{kl}P_k) \tag{A.18}$$

This rewriting of the master equation describes the change in state k of P due to the movement of probability mass from other states into k and the movement of probability mass out of k to other states. It is the discrete form of equation A.12 where P is the discretised and transformed function, ρ' with discretisation bins l and k. Q_{lk} is the proportion of neurons with a state in bin l that, after receiving a spike, move to state k.

A.3.1 Solving the Master Equation in Code

Equation A.15 shows the jump-rate matrix, $\mathbf{Q}$, for a counter of Poisson distributed events which can be modelled as a Markov chain with each state corresponding to an integer value. For discrete time, the probability of transitioning from each state to the next is equal to the rate parameter, λ, multiplied by the time step, dt. The rate parameter and time step can be factored out leaving a matrix, $\mathbf{R}$, which represents the change to each state after one event. Equation A.14 becomes equation A.19.

$$\frac{d\mathbf{P}(t)}{dt} = \lambda \mathbf{P}(t)\mathbf{R} \tag{A.19}$$

In the case where the distribution is the discretised form of ρ', the matrix $\mathbf{R}$ represents the change in probability mass after a single event, an incoming spike. $\mathbf{R}$ is an NxN matrix given that ρ' has N discrete bins. Papers A and C explain in detail the steps for generating $\mathbf{R}$. Unlike the implicit transformations described in the previous section, solving equation A.19 requires the implementation of a time-stepping algorithm. If the Euler method is used (which it is in the GPGPU implementation of MIIND), dt must be chosen so that the solution remains stable with $dt\lambda\mathbf{R}_k < 1$ for all $\mathbf{R}_k$.

Appendix B

Supplementary Material for Chapter 2

B.1 MIIND Directory Structure

Listing B.1: The main top level files in the MIIND repository.

```
MIIND_HOME/
    apps/
    cmake-modules/
    doc/
    examples/
    images/
    libs/
    package/
    Publication/
    python/
    CMakeLists.txt
    setup.py
    vcpkg.json
```

Listing B.1 shows the top-level directory structure of the MIIND code base. *libs* holds the main library code. *apps* contains the ancillary executables such as MatrixGenerator and Projection as well as various tests and C++ examples. *python* contains the full MIIND Python API including the command line interface. *examples* contains a selection of example simulations which can be run using the *miind.run* module or Python. *package* contains the files necessary to generate a Debian package and docker image of MIIND. *CMakeLists.txt* is the highest level cmake file for building MIIND, supported by additional scripts in *cmake-modules*. *setup.py* is the Python script for building and installing the MIIND Python package on all platforms. The Python installation on Windows requires the github submodule vcpkg. Finally, *doc*, *images*, and *Publication* hold files required for building the doxygen documentation.

B.2 MIIND Library Dependencies

This section lists some of the third-party libraries on which MIIND depends and are widely available on multiple platforms. The Python pip packages are designed to be distributed with all requisite dependencies to eliminate the need for any additional installation. However, for standalone functionality: generating, building and running C++ executables, these libraries must be installed in order for MIIND to function. While most are required, some libraries such as ROOT and CUDA are optional but will limit functionality if unavailable. The full list of dependencies is available in the MIIND documentation.

Root (Optional)

Root (Brun and Rademakers, 1997) is a library which provides useful data analysis and plotting tools for scientific applications. Because Root is not required for any of the major MIIND functionality, it is an optional dependency and can be disabled when building MIIND using the **ENABLE_ROOT** flag. Disabling ROOT removes the ability to use the *<SimulationIO>* block in the XML file for GeomAlgorithm. Root is disabled for the MIIND Python installation.

Boost

The basic installation of Boost is a "header-only" extension of the C++ standard. MIIND uses Boost to support parallel processing tasks such as the MPI functionality and for solving the Poisson master equation. It is also used for timing and other convenience methods. Because MIIND uses Boost for the MPI processing, the pre-built boost-mpi library must be installed before MIIND will function even if MPI is disabled.

CUDA (Optional)

In order to use the GPGPU implementation of the MIIND population density technique (MeshAlgorithmGroup and GridAlgorithmGroup), the machine running MIIND must have a CUDA enabled graphics card with the requisite CUDA libraries and drivers installed. Set **ENABLE_CUDA** in the MIIND installation to use the vectorised (Group) algorithms. CUDA is enabled in the Linux and Windows Python packages. CUDA is no longer supported on MacOS.

MPI (Optional)

MPI can be enabled in the MIIND installation with **ENABLE_MPI**. It must be supported by an installed MPI implementation such as OpenMPI or MPICH. MPI is disabled in all Python packages.

OpenMP (Optional)

OpenMP is another parallel processing solution which runs specified code blocks in parallel across multiple cores. It is recommended that OpenMP is enabled if possible even if the vectorised algorithms are to be used on the GPU. This is because OpenMP is also used to improve the performance of the geometric transition matrix generator. The **ENABLE_OPENMP** flag in the MIIND installation can be used to enable OpenMP. OpenMP is enabled for the MIIND Python package on all platforms.

B.3 CUDA Implementation

The MIIND CUDA implementation performs similar steps to the CPU version but relies on a different code path which uses the VectorisedNetwork class to contain a simulation. The MIIND Python module, *miindsimv*, instantiates (via the *init* function) VectorisedNetwork which is in the MiindLib library. VectorisedNetwork is designed to run a simulation of vectorised population densities only. For each Node defined in the simulation XML, the main class calls addGridNode, addMeshNode or addRateNode depending on the algorithm type. This is a departure from the CPU version which instantiates the algorithms themselves and passes those as parameters to each node.

The connections between nodes are defined using the functions addGridConnection, addMeshConnection and addMeshCustomConnection which instantiate an internal class for each connection type required for vectorised simulations.

Once all nodes and connections are set up, the initOde2DSystem and setupLoop functions are called which organise the separate probability mass vectors and transition matrices of the individual nodes into contiguous data structures (Fig. B.1) ready for parallel processing on the GPU. In initOde2DSystem, the probability mass vectors of each Grid and Mesh node are concatenated into a single vector, _vec_mesh. The reversal and reset mappings for each node and a list of refractive times are also generated. These structures are passed to an instance of CudaOde2DSystemAdapter in the CudaTwoDLib library. The transition matrices for the deterministic dynamics of the grid nodes are concatenated into the _csrs vector which will hold all transition matrices required for the simulation. In the setupLoop function, _csrs is further appended with the transition matrices for solving the master equation for each node (both grid and mesh). The _csrs vector along with the total number of connections, timestep and index lists (mapping the network connections to their associated transition matrices) are passed to an instance of CSRAdapter. The two classes CSRAdapter and CudaOde2DSystemAdapter in the CudaTwoDLib library are responsible for making calls to the Cuda functions defined in CudaEuler. The function singleStep in VectorizedNetwork, performs a single step of the simulation in a similar fashion to the CPU version of the code. The following algorithm steps are performed.

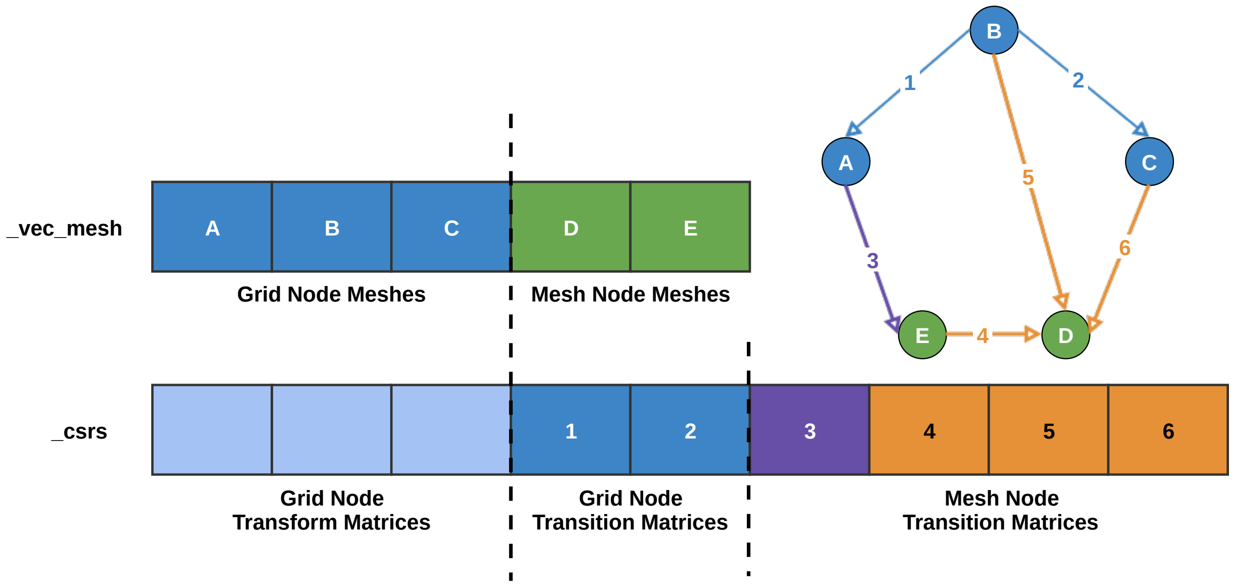

_vec_mesh
A
B
C
D
E
Grid Node Meshes
Mesh Node Meshes
_csrs
1
2
3
4
5
6
Grid Node Transform Matrices
Grid Node Transition Matrices
Mesh Node Transition Matrices

Figure B.1: The data structures used for the vectorised form of MeshAlgorithm and GridAlgorithm. Nodes A, B, and C are GridAlgorithm nodes. Nodes D and E are MeshAlgorithm nodes. _vec_mesh holds the discretised probability density functions for the grid nodes and the mesh nodes. _csrs holds the transition matrices which produce the deterministic dynamics of the grids and the matrices required to solve the Poisson master equation for each connection in the network.

B.3.1 Update the firing rates and Evolve the Deterministic Dynamics

Each connection has an associated queue of length one or more which allows for transmission delays (implemented in the same way as a refractory period as described in section 2.4 of the main text). The first task of singleStep is to pop the firing rate from the end of each queue to be passed to the associated target populations. As with the algorithm in TwoDLib::MeshAlgorithm, the deterministic dynamics of each node are progressed forward by one-time step. By calling

CudaOde2DSystemAdapter::Evolve(), the mapping of each cell in each Mesh Node, defined in TwoDLib::Ode2DSystemGroup, is shifted by one along each strip. This mapping is then passed to the graphics card. CudaOde2DSystemAdapter::RemapReversal() calls a serial cuda function defined in CudaEuler and transfers mass according to the reversal mapping which was loaded onto the card earlier.

In order to evolve the deterministic dynamics of the grid nodes, the Grid Node Transform Matrices in the _csrs vector must be applied once to their respective mass vectors.

B.3.2 Performing the Reset Remapping

In both the Grid and Mesh nodes, cells which lie on the membrane potential threshold are identified and, during each time step, mass in these cells must be moved to cells which lie on the reset potential. However, if the underlying neuron model includes a refractory period, this must also be taken into account when resetting the mass. The function, CudaOde2DSystemAdapter::RedistributeProbability(), performs the full process on the graphics card. Fig. B.2 shows the memory structure required to perform the reset functionality on the card. _res_to_mass stores the mass in cells at the threshold in the current time step. _res_sum stores the result of summing _res_to_mass and is extracted to the CPU to produce the current average firing rate. The _refractory_mass block holds the refractory queue for each threshold cell. Each refractory queue has length equal to the number of time steps in the refractory period plus one.

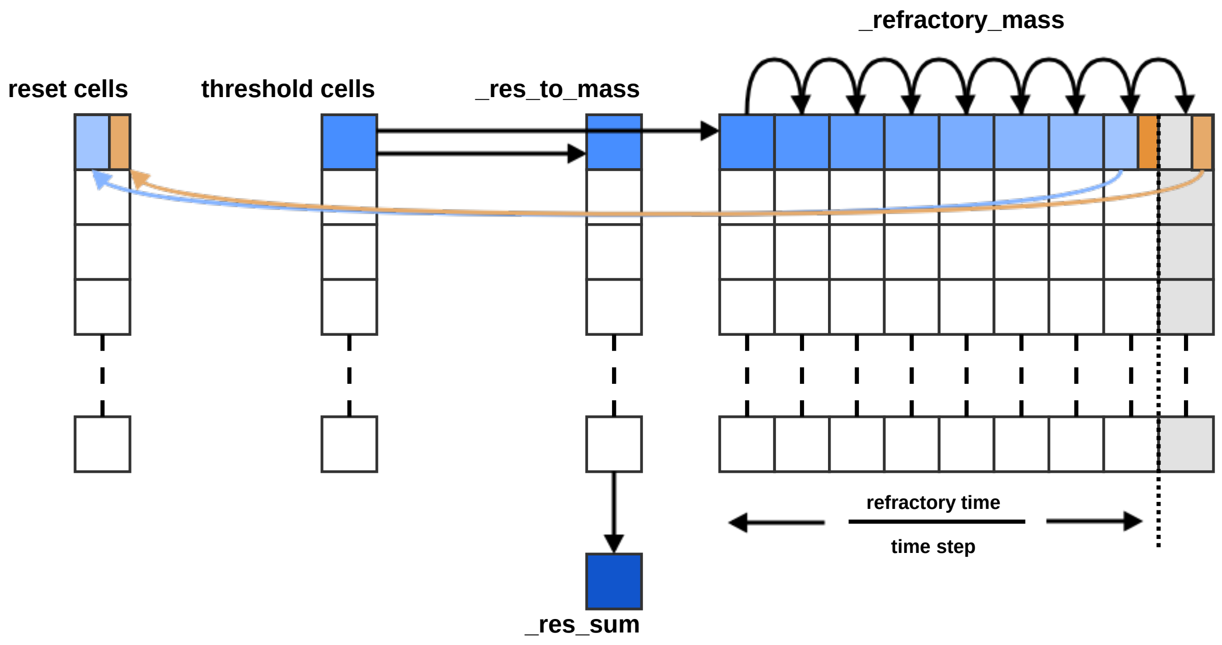


Figure B.2: The vectorised implementation of reset functionality is very similar to that of the CPU version. However, there is an additional data structure, _res_to_mass, which initially holds a copy of the mass in the threshold cells after each iteration. The structure is used to sum the cells in a parallel manner and the result is stored in _res_sum which can then be extracted from the GPU and used to calculate the average firing rate for each population.

In CudaOde2DSystemAdapter::RedistributeProbability(), first the two blocks, _res_to_mass and _res_sum, are zeroed. Next, each value in the refractory queues of _refractory_mass is shifted to the right towards the end of the queue, leaving an empty space at the beginning. CudaEuler::MapResetToRefractory() moves the mass in each threshold cell to the now-empty first position in its respective refractory queue. The threshold cell mass values are also copied to _res_to_mass for summing later. The additional position in each refractory queue is required in the case where the refractory period is not an integer multiple of the time step. In this case, only a proportion of the mass at the end of the queue is passed to the required reset cell. The remaining mass is then shifted to the additional cell and passed to the reset cell during the next iteration. Therefore, in each iteration, a reset cell receives a proportion of mass from the end cell of the queue and $1 - proportion$ of the mass from the additional cell. Finally, sumReset is called and the result of summing _res_to_mass is passed to _res_sum.

B.3.3 Solve the Master Equation for the Stochastic Noise Process

For all nodes, using both the grid and mesh algorithms, solving the spread of mass across the state space due to incoming Poisson distributed spikes is achieved through multiple applications of the associated transition matrices. The mass vector for each of the nodes in _vec_mesh is multiplied by each of the transition matrices in _csrs corresponding to incoming connections.

The VectorizedNetwork variable _master_steps, which can be set by the user in the XML file, defines how many times the matrix multiplication is performed. _master_steps must be large enough to keep the Euler numerical integration stable. The stability is dependent on the input rate of the connection and the size of the transition for each cell (which in turn is dependent on the efficacy and local dynamics of the mesh).

For this part of the process, the grid and mesh algorithms only differ in the way that the transition matrices are generated. As discussed, the mesh algorithm *.mat* files are pre-generated and passed as a static data structure. For the grid algorithm, the transitions matrix can be generated at simulation time due to the regularity of the grid cells which allows for more flexibility in their definition.

B.4 Grid Algorithm Specialisations

The regularity of the grid cells in the grid algorithm allows MIIND to calculate the transition matrix for solving the Poisson master equation at run time. When using GridAlgorithms, then, the connections can be somewhat dynamic, for example, depending on the average membrane potential of the In or Out populations. Because the way that the transition matrix is calculated can be very model specific, this must be defined in a C++ extension to the existing code base. However, two model cases are available for use "as is" or as a template. Example simulations of both of these models are available in the model archive in the code repository.

B.4.1 Minimal Motor Neuron

The minimal motor neuron model by Booth and Rinzel Booth and Rinzel, 1995 is a two-compartment model with a soma and dendrite part. Each part is comprised of a two-dimensional dynamical system including a term which integrates the membrane potential from the other compartment.

An approximation of a population of these motor neurons can be simulated in MIIND by using the specialised algorithm GridSomaDendriteAlgorithm to simulate the behaviour of a population of somas and a population of dendrites. This is clearly different to a population of individually coupled pairs which is why it is an approximation. To account for the term in each of the compartments representing the membrane potential of the opposing part, MIIND uses the average membrane potential of each population. Two connections are defined between the GridSomaDendriteAlgorithm populations to pass the shared average membrane potential values. The connections are identified with a type attribute equal to “SomaDendrite” and a “conductance” which represents the degree of coupling of the two populations as described by Booth and Rinzel, 1995. A specialised solver for the Poisson master equation solver, MasterGridSomaDendrite takes the conductance value of the connection and multiplies this by the difference between the membrane potential of each cell and the average membrane potential of the other population (provided via the connection). The GridSomaDendriteAlgorithm is currently not available for use with the GPGPU.

B.4.2 2D Epileptor

The 2D Epileptor neuron model (Proix, Bartolomei, Chauvel, et al., 2014; Proix, Bartolomei, Guye, et al., 2017) is a reduced version of the full Epileptor model (Jirsa et al., 2014) developed for use with The Virtual Brain for investigating the propagation of epileptic seizures across the whole brain. As with the GridSomaDendriteAlgorithm, the model calls for populations connected using more than just the average firing rate or membrane potential. The efficacies of two Epileptor populations are calculated using the K and tau parameters which must be provided as attributes to connections with type “epileptor”. How these values are used is described by Proix, Bartolomei, Chauvel, et al., 2014. Unlike the GridSomaDendriteAlgorithm, a separate algorithm type is not defined for the Epileptor as the integration of the population is no different to any other grid. The difference is only required at the level of the connections which must be set with the correct attributes.

B.5 Generating Marginal Density Plots

The density files generated by MeshAlgorithm and GridAlgorithm (and variants) can be used to plot both the joint distribution of the two time-dependent variables and the marginal densities showing the distribution across each individual dimension. The marginal density is essentially a histogram of the probability mass along each axis of the state space. The required dimension is split into equal-sized bins and, at a given simulation time, the total mass inside each bin is summed. For densities from MeshAlgorithm, mesh cells may span multiple bins so the correct proportion of probability mass in each cell must be attributed to the correct bin. This is also true of the GridAlgorithm although the cells are more regular. In order to calculate the attribution of mass to bins, yet again, the geometric method for generating transition matrices can be used. The area of overlap of a cell with each bin is calculated as described in section 2.3.1 of the main text and used to define the proportion to be attributed to that bin. The *Projection* program which is a separate executable similar to *GenerateMatrix* implements the algorithm for calculating the transition matrix which is then stored in a *.projection* file in the working directory of the simulation XML file. The projection file is then used for calculating the specific marginal density for a specific time in the simulation. When the **plot-marginals** command is called in *miind.miindio*, if the projection file does not exist or the mesh has been modified since the projection file was last generated, the *Projection* executable is called first then the specific marginal density can be calculated for the requested time point. As with plotting the full density, the time point parameter passed to **plot-marginals** must match a density file in the output directory of the simulation which therefore requires that the simulation includes a <*Density*> entry in the <*Recording*> section of the XML for the given population node. In order to plot the marginals of a given node in the simulation, a projection file for the associated model file must have been generated.

B.6 Building a Mesh for Mesh Algorithm

This section gives more detail on how to build a mesh for use with the mesh algorithm in MIIND. Like the grid algorithm, theoretically, any model can be built in this way. However, there are a number of subtleties to building a mesh which can require a deep understanding of the required model as well as an understanding of numerical integration techniques.

To define a single strip in the mesh, the mesh developer picks two nearby points in state space, usually some distance from the volume of space where the majority of probability mass will end up during simulation. Using a finite difference solver, stepping forward from both points according to the neuron model definition until a predetermined fixed time step is reached generates two new points. Together with the starting points, these form the first quadrilateral cell of the strip. By continuing to step forward with the solver, more cells are added to the strip. The developer should use their knowledge of the neuron model to pick starting points which lead to the strip extending into the state space where the majority of mass will stay during simulation. If the model contains a stationary point, some strips will likely approach it. As they do so, the cells will get smaller. At an appropriate point, the developer should terminate the generation of each strip such that the space between the end of the strip and the stationary point is minimised while avoiding too many small cells. The smaller and more numerous the cells get, the slower the simulation will be, so this is a balance between accuracy and performance. The potential difficulties and strategies to mitigate them are discussed in great detail by de Kamps et al., 2019. By repeating this process for many starting points around the state space, many strips can be generated that cover the space and form the whole mesh. In some instances, if the dynamics approach some discontinuity or go to infinity, the solver will fail to generate more points for a strip. Due to the difference between one edge of the strip and the other, this might cause the cells to become extremely shear. In this scenario, it might be more appropriate to backwards-integrate from starting points in this region. Further advice for mesh building is listed in section B.7. The need to sometimes cleverly pick the starting points of strips makes this process difficult to fully automate. Even changes to parameters of the same neuron model can yield very different dynamics requiring a different mesh. Once all of the vertices of the mesh have been generated they must be stored in a *.mesh* file with the format in listing B.2. The order in which the points are defined from left to right for each strip defines the direction that mass moves along the strip during the simulation. All strips must be generated with the same time step which will be used in the simulation. For convenience, the two dimensions are often labelled v for the horizontal axis and h for the vertical axis although they could represent any variable.

Listing B.2: The format for mesh files.

```
#### first line is ignored
[timestep used to generate the strips]
[tab delimited v coordinates of points along the lower side of strip 0]
[tab delimited h coordinates of points along the lower side of strip 0]
[tab delimited v coordinates of points along the upper side of strip 0]
[tab delimited h coordinates of points along the upper side of strip 0]
closed
[tab delimited v coordinates of points along the lower side of strip 1]
[tab delimited h coordinates of points along the lower side of strip 1]
[tab delimited v coordinates of points along the upper side of strip 1]
[tab delimited h coordinates of points along the upper side of strip 1]
closed
...
[tab delimited v coordinates of points along the lower side of strip n]
[tab delimited h coordinates of points along the lower side of strip n]
[tab delimited v coordinates of points along the upper side of strip n]
[tab delimited h coordinates of points along the upper side of strip n]
closed
end
```

Two further files are required before MIIND can automatically perform the remaining preprocessing steps to begin simulating a population. The mesh developer should generate a *.stat* file which defines additional quadrilaterals which will contain probability mass in the simulation which does not move along any strip. This is useful for approximating mass which has settled at a stationary point in the state space. Often there is only one stationary cell and this is defined as the starting cell for the simulation, where all probability mass initially resides. The format for the stat file is demonstrated in listing B.3 which shows the definition of a single stationary square at location[0.0,0.0] with width = 1e-05 and height = 5.0. Each quadrilateral contains the v values then the h values for all four points. As many stationary cells as are required can be added with further <*Quadrilateral*> elements. The initial mass will always be in cell [0,0].

Listing B.3: The format for .stat files.

```
<Stationary>
<Quadrilateral><vline>0.0 0.0 1e-05 1e-05</vline><wline>0 5.0 5.0 0</wline></
    ↪ Quadrilateral>
</Stationary>
```

The default behaviour for mass which reaches the end of a strip is to loop back to the initial cell. Sometimes this never happens if the strip passes through a threshold. However, for strips which approach a stationary point, the desired behaviour should be for mass to leave the end of the strip and transition to the stationary cell which surrounds the stationary point. The developer must define this behaviour in the *.rev* file. This file is similar to the transition matrix files we will see later and lists coordinates of a source cell, a target cell, and then a proportion value. Intuitively, a transition from the last cell in a strip to a stationary cell should reference these two cells. However, in the MIIND simulation loop, mass is shifted down each strip before the reversal mapping is applied. This means that the mass to be transferred to the stationary cell will reside in the initial cell of the strip as illustrated in Fig. B.3. The mapping must therefore define a transition from that first cell to the stationary cell with proportion = 1.0.

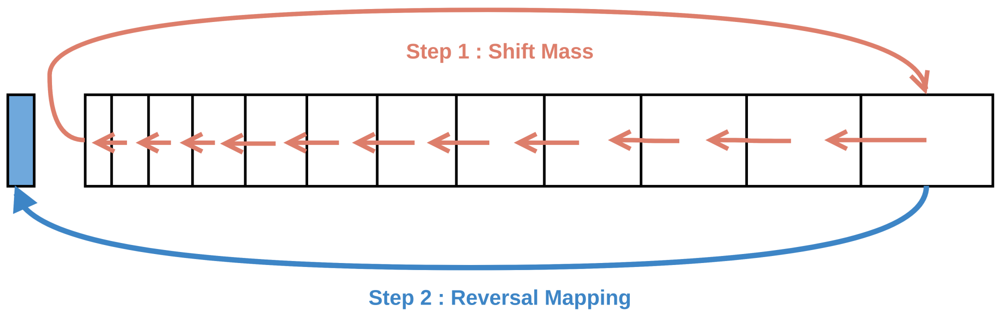


Figure B.3: Reversal mapping for a strip which approaches a stationary cell. The aim is to take probability mass from the end of the strip to the stationary cell. During MIIND's simulation loop, probability mass is first shifted one cell down the strip. The mass in the last cell is shifted to the first cell. The second step is for the reversal mapping to be applied. Therefore, the mapping should take mass from the first cell of the strip to the stationary cell.

Listing B.4 shows the format for the *.rev* file. There are other uses for the reversal file, such as when two strips are used to describe consecutive parts of a single trajectory such that when mass reaches the end of one strip it must be passed to the starting cell of the next. Another case is when one strip defines a limit cycle and approaching strips must be "stitched" so that mass can transition correctly. These scenarios are discussed in section B.7.

Listing B.4: An example reversal file which describes two transitions from the first cell in strip 1 and the first cell in strip 2 to a stationary cell at 0,0 with proportion 1.0.

```
<Mapping type="Reversal">
1,0   0,0   1.0
2,0   0,0   1.0
</Mapping>
```

The resulting three files from this process can be used by the **generate-model** command in the MIIND CLI to build a *.model* file.

B.6.1 Using the Monte Carlo Technique to Generate Transition Matrices

The main article describes how a Monte Carlo method can be used to generate transition matrices for the mesh algorithm. Points are placed in each cell and translated according to the required efficacy or jump. A proportionate transition value is given for each cell which contains one or more translated points. However, the method generates points which can be translated outside of the mesh, where there are no cells. This will happen for many cells on the edge of the mesh and for cells which are near any gaps in the mesh (for example, near a separatrix). These points must be accounted for so that the transition proportions of each cell sum to one. If a cell's transition does not sum to one, every time it is applied during the simulation, the probability mass in that cell will be removed from the system. In the event that a point falls outside the mesh, it is assigned to the cell with the nearest Euclidean distance. In order to speed up this search, the user must provide "fiducial regions" to narrow down the number of cells to test. To store these regions, an additional *.fid* file is required which can be created using the **generate-empty-fid** command in *miind.miindio*. It is initially empty but will need to be populated once the transition matrix has been generated for the first time. Although finding the closest cell is an approximation being made about the movement of mass, in many cases the gap in the mesh is small or lies along a separatrix in the state space such that a neuron in the gap would translate to its nearest cell very quickly anyway. Furthermore, the mesh itself should be designed to cover enough of the state space such that, during the simulation, large amounts of mass do not enter those cells which have transitions which extend beyond the edges. That is, the majority of mass should remain away from the edges of the mesh during simulation so any error introduced by this process affects only a very small proportion of the PDF. When **generate-matrix** is called, the lost points are recorded in the *.lost* file. If the lost file is empty, this means that there will be no lost mass during the simulation and the *.mat* file is ready to use. If the file does contain points, the user must populate the fiducial file with regions in which MIIND can search for appropriate cells to assign to each point. The **generate-matrix** command must then be run once again. *miind.miindio* provides a command, **lost**, for identifying missing points and drawing these fiducial regions. Calling **lost** with the full name of the *.lost* file will open a graphical user interface window where the user can repeatedly click the location of four corners to generate fiducial regions. The regions should be positioned so that they cover all areas of the state space where there are lost points. The edges of the regions should be far enough away from the lost points such that a new set of random points will not fall outside the edges. If the edge of a region is very close to a point, it is possible that the next time **generate-matrix** is run, that point could fall slightly outside the region and be lost again. However, the edges of the region must be close enough to the points to limit the number of cells whose distance must be checked. Fig. B.4 shows an example of fiducial regions drawn in the **lost** interface window. Once all points have been surrounded by fiducial regions, the user can double click or press enter to close the interface window and write the regions to the *.fid* file. Re-running **generate-matrix** should now yield an empty *.lost* file, guaranteeing that probability mass is conserved throughout the simulation.

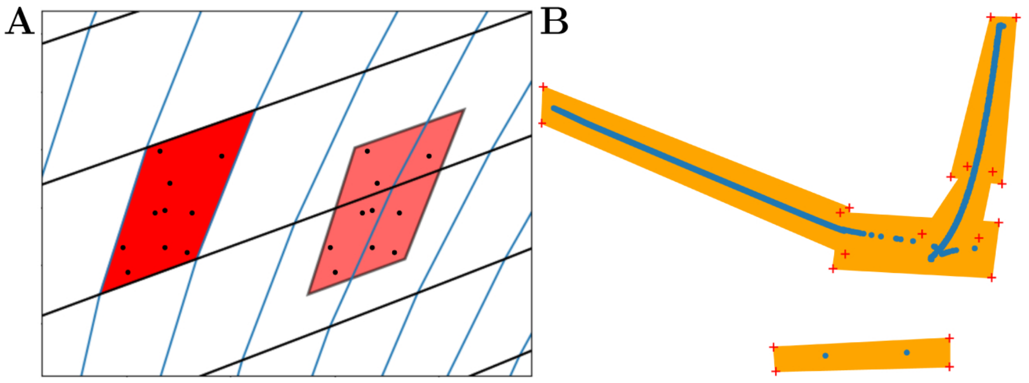


Figure B.4: (A) To calculate the transitions of each cell due to a single incoming spike, the proportion of neurons that will end up in each target cell must be calculated. The Monte Carlo method of calculating the proportions is to take a number of points randomly placed in the source cell and translate them by the efficacy amount. The geometric solution is to translate the source cell vertices themselves and calculate the area overlap with target cells. (B) When using the Monte Carlo method, cells may be translated outside of the mesh but must be accounted for. The "lost" tool displays the missing points after **generate-matrix** has been called once. The user may then draw quadrilaterals (fiducial areas) around where the points lie. During the second call to **generate-matrix**, these areas will be used to search for nearby mesh cells to which the missing points can be attributed.

B.7 Mesh Building tips

The following are a set of methods to aid in the building of meshes for use with MeshAlgorithm (see Fig. B.5 for illustrations).

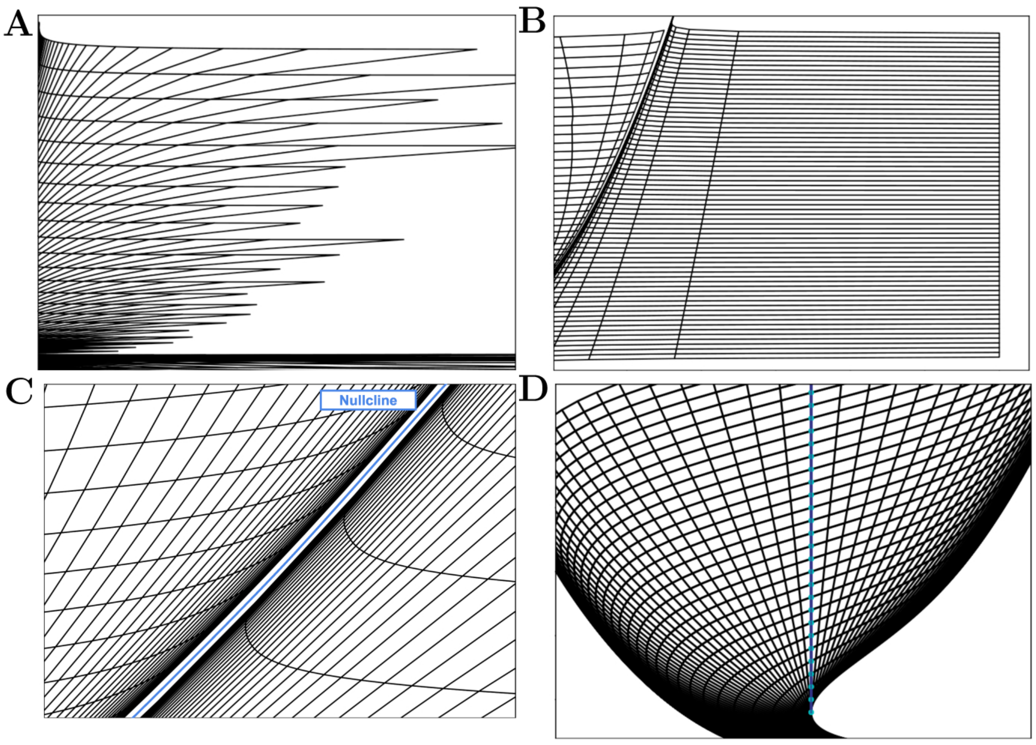


Figure B.5: (A) As trajectories moving off to the right approach infinity, each strip is cut off at a different place producing a “feathering” effect which can make it difficult to identify threshold cells. (B) By picking starting points at a high value then backwards integrating, the feathering is eliminated. (C) Picking starting points for strip trajectories on either side of a nullcline and integrating away produces only a small gap where the nullcline itself lies. (D) An example of an area of low cell density split into two by a line of starting points (in blue). This technique handles the transition from high to low-density cells on either side.

B.7.1 Backwards Integrating

Many models have a volume of state space where trajectories quickly approach infinity with the expectation that the neuron will be reset above a certain threshold. In these models, strips can become extremely shear as one side of the strip increases quicker than the other (Fig. B.5A). All points will also quickly become too high to be represented and the integration of the trajectories will fail. Sometimes it is possible to simply cut short all strips before they reach a certain threshold where these problems occur. However, an alternative is to choose the starting locations of the trajectories at some high value beyond the threshold and backwards integrate as shown in Fig. B.5B.

B.7.2 Using Nullclines

The nullclines of a model can give a good indication of where starting positions should be placed (Fig. B.5C). Where trajectories flow away from a nullcline, starting positions can be chosen just slightly on either side such that strips will travel away. The gap that is left along the nullcline itself is unimportant as the instability on that line would mean that neurons will quickly move to one side or the other. Likewise, where trajectories approach a nullcline, it can be useful to choose starting points on either side and backwards integrate.

B.7.3 Strip Stitching

Strip stitching involves using the reversal mapping to pass mass from the end of one strip to the start of another. this can be useful in a number of situations. For example, if the model to be built has a limit cycle (such as the FitzHugh-Nagumo neuron model), it is often preferable to have a single strip which follows the cycle and have strips which approach the limit cycle but stop short of the cycle strip itself. The reversal mapping can then be used to shift mass from the end of each strip to the nearest cell on the cycle strip. This prevents extreme shearing of the cells as they begin to wind around the limit cycle. Often there are areas of state space where cells are generally large and regular. Even if these are situated among areas of high cell density (for example, within the bend of a nullcline), it can be beneficial to pick a line or curve of starting points which passes through the area of low density then both forward and backwards integrate from each point. Strip stitching must be employed to attach the two opposing strips together for each of the starting points as shown in Fig. B.5D. As with reversal mappings for transferring mass to a stationary cell, the mapping must be defined between the first cell of strip one and the first cell of strip two to account for the shift of mass occurring before the mapping is performed during each time step.

B.8 Parameter Sweeps using Variables

As discussed in section 2.5.6 of the main text, the use of Variables in the XML file supports convenient methods for performing parameter sweeps. As an example, suppose that an experiment is carried out to observe the effect of noise on a population of LIF neurons. As described in section 2.10 of the main text, two quantities can be set to control the mean and variance of the change to the probability density: the input rate and the efficacy. To perform a parameter sweep across these two quantities, the appropriate variables can be declared and added to the XML as in listing B.5.

Listing B.5: An abridged version of an XML file with two variables to control the input rate and efficacy of a population of LIF neurons.

```
...
<Variable Name="RATE">1500.0</Variable>
<Variable Name="EFFICACY">0.01</Variable>
...
<Algorithm type="MeshAlgorithm" name="E" modelfile="lif.model" >
<TimeStep>0.001</TimeStep>
<MatrixFile>lif_0.01_0_0_0_.mat</MatrixFile>
<MatrixFile>lif_0.02_0_0_0_.mat</MatrixFile>
...
<MatrixFile>lif_0.1_0_0_0_.mat</MatrixFile>
</Algorithm>
<Algorithm type="RateFunctor" name="ExcInput">
<expression>RATE</expression>
</Algorithm>
<Nodes>
<Node algorithm="E" name="LIF" type="EXCITATORY_DIRECT" />
<Node algorithm="ExcInput" name="Inp" type="NEUTRAL" />
</Nodes>
<Connections>
<Connection In="Inp" Out="LIF">1 EFFICACY 0</Connection>
</Connections>
<Reporting>
...
```

When calling *miind.run* with only the XML file as a parameter, the simulation will use the default values for RATE and EFFICACY in the Variable definitions. Calling *miind.run* with additional parameters setting a name for the XML file and key-value pairs for the two variables will override the default values.

```
$ python -m miind.run lif.xml RATE=750.0 EFFICACY=0.02
```

B.8.1 Using the MIIND Python module to Perform Sweeps

An alternative to multiple *miind.run* calls is to generate a single Python script which calls *init* with the requisite variable definitions as parameters. The XML file in listing B.5 can be altered so that the rate is controlled by a Python external connection, and the efficacy remains a variable to be set in the Python script.

Listing B.6: An abridged version of an XML file to be used in a Python script with a single variable to control the efficacy.

```
...
<Variable Name="EFFICACY">0.01</Variable>
...
<Algorithm type="MeshAlgorithm" name="E" modelfile="lif.model" >
<TimeStep>0.001</TimeStep>
<MatrixFile>lif_0.01_0_0_0_.mat</MatrixFile>
<MatrixFile>lif_0.02_0_0_0_.mat</MatrixFile>
...
<MatrixFile>lif_0.1_0_0_0_.mat</MatrixFile>
</Algorithm>
<Nodes>
<Node algorithm="E" name="LIF" type="EXCITATORY_DIRECT" />
</Nodes>
<Connections>
<IncomingConnection Node="LIF">1 EFFICACY 0</IncomingConnection>
<OutgoingConnection Node="LIF"/>
</Connections>
<Reporting>
...
```

init can now be called in the Python script with the additional parameter to set the efficacy. Variable values must be passed as strings.

```
miind.init(number_of_nodes, sim_filename, EFFICACY=str(efficacy))
```

A Python program which takes the efficacy and input rate as parameters can be called multiple times and in parallel for high-performance parameter sweeps. Alternatively, the simulation can be run multiple times in the same script in a loop with different parameters. The example parameter sweep scenario shown here is available in the *examples/lif_python_sweep* directory.

B.9 Code Listings

B.9.1 Izhikevich Mesh and Grid Generators

The following MATLAB script, *izh.m*, can be used to generate a mesh file for use with a MeshAlgorithm population in MIIND. The mesh describes the dynamics of the Izhikevich Simple Model (Izhikevich, 2003) in its bursting configuration. The supporting *izhikevich.m* script defines the neuron model itself.

Listing B.7: File izh.m for generating an Izhikevich Simple Model mesh.

```
%#ok<*NOPTS>
```

```
% Reset should be -50, threshold is -30,  d = 2

revId = fopen('izh.rev', 'w');
fprintf(revId, '<Mapping Type="Reversal">\n');

outId = fopen('izh.mesh', 'w');
fprintf(outId, 'ignore\n');
fprintf(outId, '1\n');

formatSpec = '%.12f ';
strip = 0;
for u0 = -10.5:1:15.5
    tspan = 0:0.1:250;
    v0 = -150;

    [t1,s1] = ode45(@izhikevich, tspan, [v0 u0]);
    [t2,s2] = ode45(@izhikevich, tspan, [v0 u0+1]);

    x1 = s1(:,1);
    y1 = s1(:,2);

    x2 = s2(:,1);
    y2 = s2(:,2);

    svs = [];
    sus = [];
    for t = 1:min(length(x2),length(x1))
        a = x1(t);
        b = y1(t);
        c = x2(t);
        d = y2(t);

        svs = [svs, x1(t)];
        sus = [sus, y1(t)];

        plot([a c],[b d],'k');
        hold on;
    end

    fprintf(outId, formatSpec, svs);
    fprintf(outId, '\n');
    fprintf(outId, formatSpec, sus);
    fprintf(outId, '\n');

    fprintf(revId, '%i,%i\t%i,%i\t%f\n', strip, length(svs)-1, 0,0, 1.0);
    strip = strip + 1;

    axis equal
    xlim([-150 10]);
    ylim([-15 15]);

    title('Izhikevich');
    ylabel('u');
    xlabel('v');

    plot(x1,y1,'k');
    hold on;
end

fprintf(outId, 'closed\n');
%-55.5 -53.7
```

```
for v0 = -53.7:0.4:-40
    tspan = 0:0.1:200;

    eps = 0.4;

    I = 15;
    u0 = 0.04*v0^2 + 5*v0 + 140 + I + eps;

    u0_next = 0.04*(v0+0.4)^2 + 5*(v0+0.4) + 140 + I + eps;

    [t1,s1] = ode45(@izhikevich, tspan, [v0 u0]);
    [t2,s2] = ode45(@izhikevich, tspan, [v0+0.4 u0_next]);

    x1 = s1(:,1);
    y1 = s1(:,2);

    x2 = s2(:,1);
    y2 = s2(:,2);

    svs = [];
    sus = [];
    for t = 1:min(length(x2),length(x1))
        a = x1(t);
        b = y1(t);
        c = x2(t);
        d = y2(t);

        svs = [svs, x1(t)];
        sus = [sus, y1(t)];

        plot([a c],[b d],'k');
        hold on;
    end

    fprintf(outId, formatSpec, svs);
    fprintf(outId, '\n');
    fprintf(outId, formatSpec, sus);
    fprintf(outId, '\n');

    fprintf(revId, '%i,%i\t%i,%i\t%f\n', strip, length(svs)-1, 0,0, 1.0);
    strip = strip + 1;

    plot(x1,y1,'k');
    hold on;

    axis equal
    xlim([-150 10]);
    ylim([-15 15]);

    title('Izshikevich');
    ylabel('u');
    xlabel('v');
end

fprintf(outId, 'closed\n');
%-56.1  -62.0
for v0 = -62.0:1:-40
    tspan = 0:0.1:120;

    eps = 0.2;
    I = 15;
    u0 = 0.04*v0^2 + 5*v0 + 140 + I + eps;

    u0_next = 0.04*(v0+1)^2 + 5*(v0+1) + 140 + I + eps;

    [t1,s1] = ode45(@izhikevich, tspan, [v0 u0]);
    [t2,s2] = ode45(@izhikevich, tspan, [v0+1 u0_next]);

    x1 = s1(:,1);
    y1 = s1(:,2);

    x2 = s2(:,1);
    y2 = s2(:,2);

    svs = [];
    sus = [];
    for t = 1:min(length(x2),length(x1))
        a = x1(t);
        b = y1(t);
        c = x2(t);
        d = y2(t);

        svs = [svs, x1(t)];
        sus = [sus, y1(t)];

        plot([a c],[b d],'k');
        hold on;
    end

    fprintf(outId, formatSpec, svs);
    fprintf(outId, '\n');
    fprintf(outId, formatSpec, sus);
    fprintf(outId, '\n');

    fprintf(revId, '%i,%i\t%i,%i\t%f\n', strip, length(svs)-1, 0,0, 1.0);
    strip = strip + 1;

    plot(x1,y1,'k');
    hold on;

    axis equal
    xlim([-150 10]);
    ylim([-15 15]);

    title('Izhikevich');
    ylabel('u');
    xlabel('v');
end

fprintf(outId, 'closed\n');

fprintf(outId, 'end\n');

fclose(outId);

fprintf(revId, '</Mapping>\n');
fclose(revId);

outId = fopen('izh.stat', 'w');
fprintf(outId, '<Stationary>\n');
fprintf(outId, '<Quadrilateral><vline>-80.05 -80.05 -79.95 -79.95</vline><wline>4.95
    ↪ 5.05 5.05 4.95</wline></Quadrilateral>\n');
fprintf(outId, '</Stationary>\n');
fclose(outId);
```

Listing B.8: File izhikevich.m.

```
function izh = izhikevich(t, ws)
    a = 0.02;
    b = 0.2;
    I = 15;

    v = ws(1);
    u = ws(2);

    v_prime = ((0.04*v^2) + (5*v) + 140) - u + I;
    u_prime = a * ((b * v) - u);

    izsh = [v_prime; u_prime];
end
```

B.9.2 Generating the Izhikevich grid for GridAlgorithm

```
import grid_generate

def izh(y,t):
    v = y[0];
    w = y[1];

    v_prime = 0.04*v**2 + 5*v + 140 - w + 10
    w_prime = 0.02 * (0.2*v - w)

    return [v_prime, w_prime]


grid_generate.generate(izh, 0.1, 0.001, 1e-4, 'izh', -30.0, -50.0, 2.0, -85.0, -10.0,
    ↪ -15.0, 20.0, 500, 500)
```

B.9.3 Command Line Interface Command Listing

The following listing can be found by typing the command, **help**, into the command line interface provided by *miind.miindio*.

```
help                         : Get this help menu.
quit                         : Close the UI.

***** Commands for Creating and Running Simulations *****

sim      : Set the current simulation from an xml file or generate a new xml file.
models   : List all model files used by the current simulation.
settings : Set certain persistent parameters to match your MIIND installation (ENABLE
    ↪ ROOT, CUDA).
```

```
submit    : Generate and build (make) the code from the current simulation.
run       : Run the current submitted simulation.
submit-python  : Generate and build (make) a shared library for use with python from
    ↪ the current simulation.

***** Commands for Analysing and Presenting Completed Simulations *****

rate    : Plot the mean firing rate of a given node in the current simulation.
avgv    : Plot the mean membrane potential of a given node in the current simulation
plot-density    : Plot the 2D density of a given node at a given time in the
    ↪ simulation.
plot-marginals  : Plot the marginal densities of a given node at a given time in the
    ↪ simulation.
generate-density-movie  : Generate a movie of the 2D density for a given node in the
    ↪ simulation.
generate-marginal-movie : Generate a movie of the Marginal densities for a given node
    ↪  in the simulation.

***** Commands for Building New Models and Matrix Files *****

generate-model     : Generate a model file from existing mesh, rev and stat files.
generate-empty-fid : Generate a stub .fid file.
generate-matrix    : Generate a matrix file from existing model and fid files.
regenerate-reset   : Regenerate the reset mapping for an existing model.
lost               : Open the fiducial tool for capturing lost points.
generate-lif-mesh  : Helper command to build a LIF neuron mesh.
generate-qif-mesh  : Helper command to build a QIF neuron mesh.
generate-eif-mesh  : Helper command to build a EIF neuron mesh.
draw-mesh          : Draw the mesh described in an existing .mesh file.
```

References

Booth, Victoria and Rinzel, John (1995). “A minimal, compartmental model for a dendritic origin of bistability of motoneuron firing patterns”. In: *Journal of computational neuroscience* 2.4, pp. 299–312.

Brun, Rene and Rademakers, Fons (1997). “ROOT—an object oriented data analysis framework”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 389.1-2, pp. 81–86.

de Kamps, Marc, Lepperød, Mikkel, and Lai, Yi Ming (2019). “Computational geometry for modeling neural populations: From visualization to simulation”. In: *PLoS computational biology* 15.3, e1006729.

Izhikevich, Eugene M (2003). “Simple model of spiking neurons”. In: *IEEE Transactions on neural networks* 14.6, pp. 1569–1572.

Jirsa, Viktor K, Stacey, William C, Quilichini, Pascale P, Ivanov, Anton I, and Bernard, Christophe (2014). “On the nature of seizure dynamics”. In: *Brain* 137.8, pp. 2210–2230.

Proix, Timothée, Bartolomei, Fabrice, Chauvel, Patrick, Bernard, Christophe, and Jirsa, Viktor K (2014). “Permittivity coupling across brain regions determines seizure recruitment in partial epilepsy”. In: *Journal of Neuroscience* 34.45, pp. 15009–15021.

Proix, Timothée, Bartolomei, Fabrice, Guye, Maxime, and Jirsa, Viktor K (2017). “Individual brain structure and modelling predict seizure propagation”. In: *Brain* 140.3, pp. 641–654.