

# MIIND : A Model-Agnostic Simulator of Neural Populations

Hugh Osborne<sup>1</sup>, Yi Ming Lai<sup>2</sup>, Mikkel Elle Lepperød<sup>3</sup>, David Sichau<sup>4</sup>, Lukas Deutz<sup>1</sup> and Marc de Kamps<sup>1,\*</sup>

<sup>1</sup> Institute for Artificial Intelligence and Biological Computation, School of Computing, University of Leeds, Leeds, United Kingdom

<sup>2</sup> School of Medicine, University of Nottingham, Nottingham, United Kingdom

<sup>3</sup> Centre for Integrative Neuroplasticity, University of Oslo, Oslo, Norway

<sup>4</sup> Department of Computer Science, ETH Zurich, Zurich, Switzerland

Correspondence\*:

Marc de Kamps

M.deKamps@leeds.ac.uk

## 2 ABSTRACT

3 MIIND is a software platform for easily and efficiently simulating the behaviour of interacting  
4 populations of point neurons governed by any 1D or 2D dynamical system. The simulator is  
5 entirely agnostic to the underlying neuron model of each population and provides an intuitive  
6 method for controlling the amount of noise which can significantly affect the overall behaviour. A  
7 network of populations can be set up quickly and easily using MIIND's XML-style simulation file  
8 format describing simulation parameters such as how populations interact, transmission delays,  
9 post-synaptic potentials, and what output to record. During simulation, a visual display of each  
10 population's state is provided for immediate feedback of the behaviour and population activity  
11 can be output to a file or passed to a Python script for further processing. The Python support  
12 also means that MIIND can be integrated into other software such as The Virtual Brain. MIIND's  
13 population density technique is a geometric and visual method for describing the activity of each  
14 neuron population which encourages a deep consideration of the dynamics of the neuron model  
15 and provides insight into how the behaviour of each population is affected by the behaviour of its  
16 neighbours in the network. For 1D neuron models, MIIND performs far better than direct simulation  
17 solutions for large populations. For 2D models, performance comparison is more nuanced but the  
18 population density approach still confers certain advantages over direct simulation. MIIND can be  
19 used to build neural systems that bridge the scales between an individual neuron model and a  
20 population network. This allows researchers to maintain a plausible path back from mesoscopic to  
21 microscopic scales while minimising the complexity of managing large numbers of interconnected  
22 neurons. In this paper, we introduce the MIIND system, its usage, and provide implementation  
23 details where appropriate.

24 **Keywords:** Simulator, Neural Population, Population Density, Software, Python, Dynamical Systems, Network, GPU

## 1 INTRODUCTION

### 25 1.1 Population-Level Modeling

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

40 At a less granular level, rate-based techniques are a widely used practice of modeling neural activity with  
41 a single variable, whose evolution is often described by first-order ordinary differential equations, which  
42 goes back to Wilson and Cowan (1972). The Virtual Brain (TVB) uses these types of models to represent  
43 activity of large regions (nodes) in whole brain networks to generate efficient simulations (Sanz Leon et al.,  
44 2013; Jirsa et al., 2014). TVB demonstrates the benefits of a rate based approach with the Epileptor neural  
45 population model yielding impressive clinical results (Proix et al., 2017). The Epileptor model is based  
46 on the well known Hindmarsh-Rose neuron model (Hindmarsh and Rose, 1984). However, the behaviour  
47 of this and other rate based models is defined at the population level instead of behaviour emerging from  
48 a definition of the underlying neurons. Therefore, these models have less power to explain simulated  
49 behaviours at the microscopic level.

50 Between these two extremes of granularity is a research area which bridges the scales by deriving  
51 population level behaviour from the behaviour of the underlying neurons. So called population density  
52 techniques (PDTs) have been used for many years (Knight, 1972; Knight et al., 1996; Omurtag et al., 2000)  
53 to describe a population of neurons in terms of a probability density function. The transfer function of a  
54 neuron model or even an experimental neural recording can be used to approximate the response from a  
55 population using this technique (Wilson and Cowan, 1972; El Boustani and Destexhe, 2009; Carlu et al.,  
56 2020). However, analytical solutions are often limited to regular spiking behaviour with constant or slowly  
57 changing input. The software we present here, MIIND, provides a numerical solution for populations of  
58 neurons with potentially complex behaviours (for example bursting) receiving rapidly changing noisy input  
59 with arbitrary jump sizes. The noise is usually assumed to be shot noise, but MIIND can also be used  
60 with other renewal processes such as Gamma distributed input (Lai and de Kamps, 2017). It contains a  
61 number of features that make it particularly suitable for dynamical systems representing neuronal dynamics,  
62 such as an adequate handling of boundary conditions that emerge from the presence of thresholds and  
63 reset mechanisms, but is not restricted to neural systems. The dynamical systems can be grouped in large  
64 networks, which can be seen as the model of a neural circuit at the population level.

65 The key idea behind MIIND is shown in Fig. 1A. Here, a population of neurons is simulated. In this case,  
66 the neurons are defined by a conductance based leaky-integrate-and-fire neuron model with membrane

67 potential and state of the conductance as the two variables. The neuron's evolution through state space is  
 68 given by a two-dimensional dynamical system described by Eq. 1.

$$\begin{aligned}\tau \frac{dV}{dt} &= -g_l(V - E_l) - g_e(t)V, \\ \tau_e \frac{dg_e}{dt} &= -g_e + I_{syn}(t)\end{aligned}\quad (1)$$

69  $V$  is the membrane potential and  $g_e$  is the conductance variable.  $E_l$  (set to -65mV in this example) is the  
 70 reversal potential and  $\tau$  (20ms) and  $\tau_e$  (5ms) represent the time scales for  $V$  and  $g_e$  respectively.  $I_{syn}$   
 71 represents changes to the conductance variable due to incoming spikes. If  $V$  is raised above a specified  
 72 threshold value (-55mV), it is reset to a specified reset membrane potential (-65mV). The positions of  
 73 individual neurons change in state space, both under the influence of the neuron's endogenous dynamics  
 74 as determined by the dynamical system and of spike trains arriving from neurons in other populations,  
 75 which cause rapid transitions in state space that are modeled as instantaneous jumps. For the simulation  
 76 techniques mentioned earlier involving a large number of individual model neuron instances, a practice that  
 77 we will refer to as Monte Carlo simulation, the population can be represented as a cloud of points in state  
 78 space. The approach in MIIND, known as a population density technique (PDT) models the probability  
 79 density of the cloud, shown in Fig. 1 as a heat map, rather than the behaviour of individual neurons. The  
 80 threshold and reset values of the underlying neuron model are visible in the hard vertical edges of the  
 81 density in Fig. 1A. In Fig. 1B, the same simulation approach is used for a population of Fitzhugh-Nagumo  
 82 neurons (FitzHugh, 1961; Nagumo et al., 1962). The dynamical system is defined in Eq. 2.

$$\begin{aligned}\frac{dV}{dt} &= V - \frac{V^3}{3} - W, \\ \frac{dW}{dt} &= 0.08(V + 0.7 - 0.8W)\end{aligned}\quad (2)$$

83  $V$  represents the membrane potential and  $W$  is a recovery variable. The Fitzhugh-Nagumo model  
 84 has no threshold-reset mechanism and so there are no vertical boundaries to the density. As well  
 85 as being informative in themselves, common population metrics such as average firing rate and ave-  
 86 rage membrane potential can be quickly calculated from these density functions. The MIIND model  
 87 archive, available in the code repository, contains example simulation files for populations of both con-  
 88 ductance based neurons and Fitzhugh-Nagumo neurons (*examples/model\_archive/Conductance2D* and  
 89 *examples/model\_archive/FitzhughNagumo*).

## 90 1.2 The Case for Population Density Techniques

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

98 Apart from simulation speed, PDTs have been important in understanding population level behaviour  
 99 analytically. Important questions, such as 'why are cortical networks stable?' (Amit and Brunel, 1997),

100 ‘how can a population be oscillatory when its constituent neurons fire sporadically?’ (Brunel and Hakim,  
 101 1999), ‘how does spike shape influence the transmission spectrum of a population?’ (Fourcaud-Trocmé  
 102 et al., 2003) have been analysed in the context of population density techniques, providing insights that  
 103 cannot be obtained from merely running simulations. A particularly important question, which has not been  
 104 answered in full is: ‘how do rate-based equations emerge from populations of spiking neurons and when is  
 105 their use appropriate?’. There are many situations where such rate-based equations are appropriate, but  
 106 some where they are not and their correspondence to the underlying spiking neural dynamics is not always  
 107 clear (Montbrió et al., 2015; de Kamps, 2013). There is a body of work suggesting that some rate-based  
 108 equations can be seen as the lowest order of perturbations of a stationary state, and much of this work is  
 109 PDT-based (Wilson and Cowan, 1972; Gerstner, 1998; Mattia and Del Giudice, 2002, 2004; Montbrió  
 110 et al., 2015). MIIND opens the possibility to incorporate these theoretical insights into large-scale network  
 111 models. For example, we can demonstrate the prediction from Brunel and Hakim (1999) that inhibitory  
 112 feedback on a population can cause a bifurcation and produce resonance. Finally, for a steady state input,  
 113 the firing rate prediction of a PDT model converges to a transfer function which can be used in artificial  
 114 spiking neural networks (De Kamps et al., 2008).

### 115 1.3 Population-level Modeling

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

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial \vec{v}} \cdot \left( \frac{\vec{F}(\vec{v})\rho(\vec{v}, t)}{\tau} \right) = \int_M d\vec{v}' \left\{ W(\vec{v} \mid \vec{v}')\rho(\vec{v}') - W(\vec{v}' \mid \vec{v})\rho(\vec{v}) \right\}, \quad (3)$$

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

$$\tau \frac{d\vec{v}}{dt} = \vec{F}(\vec{v}), \quad (4)$$

123 where  $\tau$  is the neuron’s membrane time constant. Simple models are one-dimensional (1D). For the  
 124 leaky-integrate-and-fire (LIF) neuron:

$$F(v) = -v, \quad (5)$$

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

$$F(v) = v^2 + I, \quad (6)$$

126 where  $v$  is the membrane potential, and  $I$  can be interpreted as a bifurcation parameter. More complex  
 127 models require a higher dimensional state space. Since such a space is hard to visualise and understand,  
 128 considerable effort has been invested in the creation of effective models. In particular two-dimensional  
 129 (2D) models are considered to be a compromise that allows considerably more biological realism than LIF  
 130 or QIF neurons, but which remain amenable to visualisation and analysis, and can often be interpreted  
 131 geometrically (Izhikevich, 2007). Examples are the Izhikevich simple neuron (Izhikevich, 2003), the

132 Fitzhugh-Nagumo neuron (FitzHugh, 1961; Nagumo et al., 1962), and the adaptive-exponential-integrate-  
 133 and-fire neuron (Brette and Gerstner, 2005), incorporating phenomena such as bursting, bifurcations,  
 134 adaptation, and others that cannot be accounted for in a one dimensional model.

135  $W(v' | v)$  in Eq. 3 represents a transition probability rate function. The right hand side of Eq. 3 makes it  
 136 a Master equation. Any Markovian process can be represented by a suitable choice of  $W$ . For example, for  
 137 shot noise, we have

$$W(v' | v) = \nu(\delta(v' - v - h) - \delta(v - v')), \quad (7)$$

138 where  $\nu$  is the rate of the Poisson process generating spike events. The delta functions reflect that an  
 139 incoming spike causes a rapid change in state space, modeled as an instantaneous jump,  $h$ . It depends on the  
 140 particular neural model in what variable the jumps take place. Often models use a so-called delta synapse,  
 141 such that the jump is in membrane potential. In conductance based models, the incoming spike causes a  
 142 jump in the conductance variable (Fig. 1A), and the influence of the incoming spike on the potential is then  
 143 indirect, given by the dynamical system's response to the sudden change in the conductance state.

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

## 150 1.4 Quick Start Guide

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

159 `$ python -m miind.loadExamples`

160 In the *examples/quick\_start* directory, the *generateCondFiles.py* script generates the simulation files,  
 161 *cond.model* and *cond.tmat*.

162 `$ python generateCondFiles.py`

163 The contents of *generateCondFiles.py* is given in Listing 1. The two important parts of the script  
 164 are the neuron model function, in this case named *cond()*, and the call to the MIIND function  
 165 *grid\_generate.generate()* which takes a number of parameters which are discussed in detail later.

### Listing 1. *generateCondFiles.py*

```
166 import miind.grid_generate as grid_generate
167
168 def cond(y, t):
169     E_r = -65e-3
170     tau_m = 20e-3
171     tau_s = 5e-3
```

```

172
173     v = y[0];
174     h = y[1];
175
176     v_prime = ( -(v - E_r) - (h * v) ) / tau_m
177     h_prime = -h / tau_s
178
179     return [v_prime, h_prime]
180
181 grid_generate.generate(
182     func = cond,
183     timestep = 1e-04,
184     timescale = 1,
185     tolerance = 1e-6,
186     basename = 'cond',
187     threshold_v = -55.0e-3,
188     reset_v = -65e-3,
189     reset_shift_h = 0.0,
190     grid_v_min = -72.0e-3,
191     grid_v_max = -54.0e-3,
192     grid_h_min = -1.0,
193     grid_h_max = 2.0,
194     grid_v_res = 200,
195     grid_h_res = 200,
196     efficacy_orientation = 'h'

```

197 The *cond()* function should be familiar to those who have used Python numerical integration frameworks  
 198 such as *scipy.integrate*. It takes the two time dependent variables defined by *y*[0] and *y*[1] and a placeholder  
 199 parameter, *t*, for performing a numerical integration. In the function, the user may define how the derivatives  
 200 of each variable are to be calculated. The *generate()* function requires a suitable time step, values for a  
 201 threshold and reset if needed, and a description of the extent of the state space to be simulated. With this  
 202 structure, the user may define any two dimensional neuron model. The generated files are then referenced  
 203 in a second file which describes a network of populations to be simulated. Listing 2 shows the contents of  
 204 *cond.xml* describing an E-I network which uses the generated files from *generateCondFiles.py*.

### **Listing 2. *cond.xml***

```

205 <Simulation>
206     <WeightType>CustomConnectionParameters</WeightType>
207     <Algorithms>
208         <Algorithm type="GridAlgorithm" name="COND" modelfile="cond.model" tau_refractive="0.0"
209             ↪ transformfile="cond_0_0_0_0_.tmat" start_v="-0.065" start_w="0.0" >
210             <TimeStep>1e-04</TimeStep>
211         </Algorithm>
212         <Algorithm type="RateFunctor" name="ExcitatoryInput">
213             <expression>800.</expression>
214         </Algorithm>
215     </Algorithms>
216     <Nodes>
217         <Node algorithm="ExcitatoryInput" name="INPUT_E" type="EXCITATORY_DIRECT" />
218         <Node algorithm="ExcitatoryInput" name="INPUT_I" type="EXCITATORY_DIRECT" />
219             <Node algorithm="COND" name="E" type="EXCITATORY_DIRECT" />
220             <Node algorithm="COND" name="I" type="INHIBITORY_DIRECT" />
221     </Nodes>

```

```

222     <Connections>
223         <Connection In="INPUT_E" Out="E" num_connections="1" efficacy="0.1" delay="0.0"/>
224         <Connection In="INPUT_I" Out="I" num_connections="1" efficacy="0.1" delay="0.0"/>
225             <Connection In="E" Out="I" num_connections="1" efficacy="0.1" delay="0.001"/>
226             <Connection In="E" Out="E" num_connections="1" efficacy="0.1" delay="0.001"/>
227             <Connection In="I" Out="E" num_connections="1" efficacy="-0.1" delay="0.001"/>
228             <Connection In="I" Out="I" num_connections="1" efficacy="-0.1" delay="0.001"/>
229     </Connections>
230     <Reporting>
231         <Display node="E" />
232         <Display node="I" />
233         <Rate node="E" t_interval="0.001" />
234         <Rate node="I" t_interval="0.001" />
235     </Reporting>
236     <SimulationRunParameter>
237         <SimulationName>EINetwork</SimulationName>
238         <t_end>0.2</t_end>
239         <t_step>1e-04</t_step>
240         <name_log>einetwork.log</name_log>
241     </SimulationRunParameter>
242 </Simulation>

```

243 The full syntax documentation for MIIND XML files is given in section 4. Though more compact or  
 244 flexible formats are available, XML was chosen as a formatting style due to its ubiquity ensuring the  
 245 majority of users will already be familiar with the syntax. The *Algorithms* section is used to declare specific  
 246 simulation methods for one or more populations in the network. In this case, a GridAlgorithm named  
 247 *COND* is set up which references the *cond.model* and *cond.tmat* files. A RateFunctor algorithm produces a  
 248 constant firing rate. In the *Nodes* section, two instances of *COND* are created: one for the excitatory and  
 249 inhibitory populations respectively. Two *ExcitatoryInput* nodes are also defined. The *Connections* section  
 250 allows us to connect the input nodes to the two conductance populations. The populations are connected to  
 251 each other and to themselves with a 1ms transmission delay. The remaining sections are used to define how  
 252 the output of the simulation is to be recorded, and to provide important simulation parameters such as the  
 253 simulation time. By running the following python command, the simulation can be run.

**Listing 3.** Run the cond.xml simulation.

```
254 $ python -m miind.run cond.xml
```

255 The probability density plots for both populations will be displayed in separate windows as the simulation  
 256 progresses. The firing rate of the excitatory population can be plotted using the following commands. Fig.  
 257 2 shows the probability density plots for both populations and average firing rate of population E.

**Listing 4.** Load the cond.xml simulation and plot the average firing rate of population E.

```
258 $ python -m miind.miindio sim cond.xml
259 $ python -m miind.miindio rate E
```

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

**Listing 5.** Plot the probability density of population I at time 0.12s.

```
262 $ python -m miind.miindio plot-density I 0.12
```

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

## 2 THE MIIND GRID ALGORITHM

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

### 2.1 Generating the Grid and Transition Matrix

280 To discretise the state space in the grid method, the user can specify the size and  $M \times N$  resolution of  
 281 a rectangular grid which results in  $MN$  identical rectangular cells, each of which will hold probability  
 282 mass. In the grid algorithm, a transition matrix lists the proportion of mass which moves from each cell to  
 283 (usually) adjacent cells in one time step due to the deterministic dynamics of the underlying neural model.  
 284 To pre-calculate the transitions for each cell, MIIND first translates the vertices of every cell by integrating  
 285 each point forward by one time step according to the dynamics of the underlying neuron model as shown  
 286 in Fig. 3A. As the time step is small, a single Euler step is usually all that is required to avoid large errors  
 287 (although other integration schemes can be used if required). Each transformed cell is no longer guaranteed  
 288 to be a rectangle and is compared to the original non-transformed grid to ascertain which cells overlap with  
 289 the newly generated quadrilateral. An overlap indicates that some proportion of neurons in the original  
 290 cell will move to the overlapping cell after one time step. In order to calculate the overlap, the algorithm  
 291 in Listing 6 is employed. This algorithm is also used in the geometric method of generating transition  
 292 matrices for the mesh algorithm shown later.

**Listing 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.

293 For each transformed cell, A, in the grid:  
 294     Translate all four vertices according to a single Euler step.  
 295     Split A into two triangles and add them to a triangle list.  
 296 For each non-transformed cell, B:  
 297     Set the overlapping area sum to 0.  
 298     While the triangle list has changed:  
 299         For each triangle in the list:  
 300             If the triangle is entirely outside B: add 0 to the sum.  
 301             If the triangle is entirely within B: add the triangle's area to the sum.  
 302             If B is entirely within the triangle: add B's area to the sum.  
 303             Else: For each edge in B:

---

304     Calculate any intersection points with the edges of the triangle.  
 305     Triangulate the polygon produced by the original triangle points plus the new intersection  
 306         → points.  
 307     Remove the original triangle from the list.  
 308     Add the newly generated triangles to the list.  
 309     Calculate the proportion of A taken by the sum.  
 310     Add the transition from A to B with the proportion to the transition matrix.

311     Though the pseudo-code algorithm is order  $N^2$ , there are many ways that the efficiency of the algorithm  
 312 is improved in the implementation. The number of non-transformed cells checked for overlap can be limited  
 313 to only those which lie underneath each given triangle. Furthermore, the outer loop is parallelisable. Finally,  
 314 as the non-transformed cells are axis-aligned rectangles, the calculation to find edge intersections is trivial.  
 315 Fig. 3A shows a fully translated and triangulated cell at the end of the algorithm. Once the transition matrix  
 316 has been generated, it is stored in a file with the extension *.tmat*. Although the regular grid can be described  
 317 with only four parameters (the width, height, X, and Y resolutions), to more closely match the behaviour of  
 318 mesh algorithm, the vertices of the grid are stored in a *.model* file. To simulate a population using the grid  
 319 algorithm, the *.tmat* and *.model* files must be generated and referenced in the XML simulation file.

320     As demonstrated in the quick start guide (section 1.4), to generate a *.model* and *.tmat* file, the user must  
 321 write a short Python script which defines the underlying neuron model and makes a call to the MIIND API  
 322 to run the algorithm in listing 6. In the *python* directory of the MIIND source repository (see section 1  
 323 in the supplementary material), there are a number of examples of these short scripts. The script used to  
 324 generate a grid for the Izhikevich simple model is listed in the supplementary material section 9.1. The  
 325 required definition of the neuron model function is similar to those used by many numerical integration  
 326 libraries. The function takes a parameter, *y*, which represents a list which holds the two time dependent  
 327 variables and a parameter, *t*, which is a placeholder for use in integration. The function must return the first  
 328 time derivatives of each variable as a list in the same order as in *y*. Once the function has been written, a  
 329 call to *grid\_generate.generate* is made which takes the parameters listed in Table 1.

330     When the user runs the script, the required *.model* and *.tmat* files will be generated for use in a simulation.  
 331 In the quick start guide, the conductance based neuron model requires that *efficacy\_orientation* is set to  
 332 ‘h’ because incoming spikes cause an instantaneous change in the conductance variable instead of the  
 333 membrane potential. By default, however, this parameter is set to ‘v’. When choosing values for the grid  
 334 bounds (*grid\_v\_min*, *grid\_v\_max*, *grid\_h\_min*, and *grid\_h\_max*), the aim is to estimate where in state space  
 335 the population density function might be non-zero during a simulation. In the conductance based neuron  
 336 model, because of the threshold-reset mechanic, the *grid\_v\_max* parameter need only be slightly above  
 337 the threshold to ensure that there is at least one column of cells on or above threshold to allow probability  
 338 mass to be reset. The *grid\_v\_min* value should be below the resting potential and reset potential. However,  
 339 we must also consider that the neurons could receive inhibitory spikes which would cause the neurons to  
 340 hyperpolarise. *grid\_v\_min* should therefore be set to a value beyond the lowest membrane potential expected  
 341 during the simulation. Similarly for the conductance variable, space should be provided for reasonable  
 342 positive and negative values. If it is known beforehand that no inhibition will occur, however, then the state  
 343 space bounds can be set tighter in order to improve the accuracy of the simulation using the same grid  
 344 resolution (*grid\_v\_res* and *grid\_h\_res*). If, during the simulation, probability mass is pushed beyond the  
 345 lower bounds of the grid, it will be pinned at those lower bounds which will produce incorrect behaviour  
 346 and results. If the probability mass is pushed beyond the upper bounds, it will be wrapped around to the  
 347 lower bounds which will also produce incorrect results. The choice of grid resolution is a balance between  
 348 speed of simulation and accuracy. However, even very coarse grids can produce representative firing rates

349 and behaviours. Typical grid resolutions range between 100x100 and 500x500. It can also be beneficial  
 350 to experiment with different  $M$  and  $N$  values as the accuracy of each dimension can have unbalanced  
 351 influence over the population level metrics.

352

## 353 2.2 The Effect of Random Incoming Spikes

354 The transition matrix in the *.tmat* file describes how probability mass moves to other cells due to the  
 355 deterministic dynamics of the underlying neuron model. The transition matrix is sparse as probability mass  
 356 is often only transferred to nearby cells. Solving the deterministic dynamics is therefore very efficient. The  
 357 mesh algorithm is even faster and, as demonstrated later, is significantly quicker than direct simulation  
 358 for this part of the algorithm. Another benefit to the modeler is that by rendering the grid with each cell  
 359 coloured according to its mass, the resultant heat map gives an excellent visualisation of the state of the  
 360 population as a whole at each time step of the simulation as shown in Fig. 2. This provides particularly  
 361 useful insight into the sub-threshold behaviour of neurons in the population.

362 The second step of the grid algorithm, which must be performed every iteration, is to solve the change  
 363 in the probability density function due to random incoming spikes. It is assumed that a spike causes an  
 364 instantaneous change in the state of a neuron, usually a step wise jump in membrane potential corresponding  
 365 to a constant synaptic efficacy. In the conductance based neuron example, this jump is in the conductance.  
 366 When considering each cell in the grid, a single incoming spike will cause some proportion of the  
 367 probability mass to shift to at most, two other cells as shown in Fig. 3. Because all cells in the grid are  
 368 equally distributed and the same size, the relative transition of probability mass caused by a single spike is  
 369 the same for them all. A sparse transition matrix,  $M$ , can be generated from this single transition so that  
 370 applying  $M$  to the probability density grid applies the transition to all cells. MIIND calculates a different  
 371  $M$  for each incoming connection to the population based on the user defined instantaneous jump, which  
 372 we refer to as the efficacy. In the mesh algorithm, the relative transitions are different for each cell and so a  
 373 transition matrix (similar to that of the *.tmat* file) is required to describe the effect of a single spike. As with  
 374 many other population density techniques, MIIND assumes that incoming spikes are Poisson distributed,  
 375 although it is possible to approximate other distributions. MIIND uses  $M$  to calculate the change to the  
 376 probability density function,  $\rho$ , due solely to the non-deterministic dynamics as described by equation 8.

$$d\rho/dt = \lambda M \rho \quad (8)$$

377  $\lambda$  is the incoming Poisson firing rate. The boost numeric library is used to integrate  $d\rho/dt$ . The solution  
 378 to this equation describes the spread of the probability density due to Poisson spikes. This ‘master process’  
 379 step amounts to multiple applications of the transition matrix  $M$  and is where the majority of time is taken  
 380 computationally. However, OpenMP is available in MIIND to parallelise the matrix multiplication. If  
 381 multiple cores are available, the OpenMP implementation significantly improves performance of the master  
 382 process step. More information covering this technique can be found in De Kamps et al. (2019); de Kamps  
 383 (2013).

384

## 385 2.3 Threshold-Reset Dynamics

386 Many neuron models include a “threshold-reset” process such that neurons which pass a certain mem-  
 387 brane potential value are shifted back to a defined reset potential to approximate repolarisation during an  
 388 action potential. To facilitate this in MIIND, after each iteration, probability mass in cells which lie across  
 389 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. 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.

400

## 401 2.4 How MIIND Facilitates Interacting Populations

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

417 The simplicity of the XML file means that a user can set up a large network of populations with very  
418 little effort. The model archive in the code repository holds a set of example simulations demonstrating the  
419 range of MIIND’s functionality and includes an example which simulates the Potjans-Diesmann model  
420 of a cortical microcircuit (Potjans and Diesmann, 2014), which is made up of eight populations of leaky  
421 integrate and fire neurons. Fig. 5 shows a representation of the model with embedded density plots for each  
422 population.

423

## 424 2.5 Running MIIND Simulations

425 The quick start guide demonstrated the simplest way to run a simulation given that the required *.model*,  
426 *.tmat*, and *.XML* files have been generated. The *miind.run* script imports the *miind.miindsim* Python  
427 extension module which can also be imported into any user written Python script. Section 6 details the  
428 functions which are exposed by *miind.miindsim* for use in a python script. The benefit of this method is  
429 that the outputs from populations can be recorded after each iteration and inputs can be dynamic allowing  
430 the python script to perform its own logic on the simulation based on the current state.

431 There is also a command line interface (CLI) program provided by the Python module, *miind.miindio*.  
432 The CLI can be used for many simple work flow tasks such as generating models and displaying results.  
433 Each command which is available in the CLI, can also be called from the MIIND Python API, upon which

434 the CLI is built. A full list of the available commands in the CLI is given in section 9.3 of the supplementary  
435 material and a worked example using common CLI commands is provided in section 7.

436

## 437 2.6 When not to use the Grid Algorithm

438 For many underlying neuron models, the grid algorithm will produce results showing good agreement  
439 with direct simulation to a greater or lesser extent depending on the resolution of the grid (see Fig. 6).  
440 However, for models such as exponential integrate and fire, a significantly higher grid resolution is required  
441 than might be expected because of the speed of the dynamics across the threshold (beyond which, neurons  
442 perform the action potential). When the input rate is high enough to generate tonic spiking in an exponential  
443 integrate and fire model, the rate of depolarisation of each neuron reduces as it approaches the threshold  
444 potential then once it is beyond the threshold, quickly increases producing a spike. Because the grid  
445 discretises the state space into regular cells, if cells are large due to a low resolution, only a small number  
446 of cells will span the threshold, as shown in Fig. 7A. When the transition matrix is applied each time  
447 step, probability mass is distributed uniformly across each cell. Probability mass can therefore artificially  
448 cross the threshold much faster than it should leading to a higher than expected average firing rate for the  
449 population. Using the grid algorithm for such models where the firing rate itself is dependent on sharp  
450 changes in the speed of the dynamics should be avoided if high accuracy is required. Other neuron models,  
451 like the bursting Izhikevich simple model, also have sharp changes in speed when neurons transition from  
452 bursting to quiescent periods. However, the bursting firing rate is unaffected by these dynamics and the  
453 oscillation frequency is affected only negligibly due to the difference in timescales. The grid algorithm  
454 is therefore still appropriate in cases such as this. For exponential integrate and fire models, however,  
455 MIIND provides a second algorithm which can more accurately capture the deterministic dynamics: mesh  
456 algorithm.

## 3 THE MIIND MESH ALGORITHM

457 Instead of a regular grid to discretise the state space of the underlying neuron model, the mesh algorithm  
458 requires a two dimensional mesh which describes the dynamics of the neuron model itself in the absence  
459 of incoming spikes. A mesh is constructed from strips which follow the trajectories of neurons in state  
460 space (Fig. 8). The trajectories form so-called characteristic curves of the neuron model from which this  
461 method is inspired (De Kamps et al., 2019; de Kamps, 2013).

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

472 As with the grid algorithm, when the simulation is running, each cell is associated with a probability  
473 mass value which represents the probability of finding a neuron from the population with a state in that  
474 cell. When a probability density function (PDF) is defined across the mesh, computing the change to the  
475 PDF due to the deterministic dynamics of the neurons is simply a matter of shifting each cell’s probability

476 mass value along its strip. In the C++ implementation, this requires no more than a pointer update and is  
477 therefore quicker than the grid algorithm for solving the deterministic dynamics as no transition matrix is  
478 applied to the cells.

479 Mesh algorithm does, however, still require a transition matrix to implement the effect of incoming spikes  
480 on the PDF. This transition matrix describes how the state of neurons in each cell are translated in the event  
481 of a single incoming spike. Unlike the grid algorithm, cells are unevenly distributed across the mesh and  
482 are different sizes and shapes. What proportion of probability mass is transferred to which cells with a  
483 single incoming spike is, therefore, different for all cells. During simulation, the total change in the PDF is  
484 calculated by shifting probability mass one cell down each strip and using the transition matrix to solve the  
485 master equation every time step. The combined effect can be seen in Fig. 9. The method of solving the  
486 master equation is explained in detail in de Kamps (2013).

### 487 3.1 When not to use the Mesh Algorithm

488 Just as with the grid algorithm, certain neuron models are better suited to an alternative algorithm. In  
489 the mesh algorithm, very little error is introduced for the deterministic dynamics. Probability mass flows  
490 down each strip as it would without the discretisation and error is limited only to the size of the cells.  
491 When the master equation is solved, however, probability mass can spread to parts of state space which  
492 would see less or no mass. Fig. 7B demonstrates how in the mesh algorithm, as probability mass is pushed  
493 horizontally, very shear cells can allow mass to be incorrectly transferred vertically as well. In the the  
494 grid algorithm, error is introduced in the opposite way. Solving the master equation pushes probability  
495 mass along horizontal rows of the grid and error is limited to the width of the row. The grid algorithm is  
496 preferable over the mesh algorithm for populations of neurons with one fast variable and one slow variable  
497 which can produce very shear cells in a mesh, e.g. in the Fitzhugh-Nagumo model (De Kamps et al., 2019).  
498 In both algorithms, the error can be reduced by increasing the density of cells (by increasing the resolution  
499 of the grid, or by reducing the timestep and strip width of the mesh). However, better efficiency is achieved  
500 by using the appropriate algorithm.

### 501 3.2 Building a Mesh for the Mesh Algorithm

502 Before a simulation can be run for a population which uses the mesh algorithm, the pre-calculation steps  
503 of generating a mesh and transition matrices must be performed. Fig. 10 shows the full pre-processing  
504 pipeline for mesh algorithm. The mesh is a collection of strips made up of quadrilateral cells. As mentioned  
505 earlier, probability mass moves along a strip from one cell to the next each time step which describes  
506 the deterministic dynamics of the model. Defining the cells and strips of a 2D mesh is not generally a  
507 fully automated process and the points of each quadrilateral must be defined by the mesh developer and  
508 stored in a *.mesh* file. When creating the mesh, the aim is to cover as much of the state space as possible  
509 without allowing cells to get too small or misshapen. An example of a full mesh generation script for  
510 the Izhikevich simple neuron model (Izhikevich, 2003) is available in section 9.1 of the supplementary  
511 material. MIIND provides *miind.miind.api.LifMeshGenerator*, *miind.miind.api.QifMeshGenerator*, and  
512 *miind.miind.api.EifMeshGenerator* scripts to automatically build the 1D leaky integrate and fire, quadratic  
513 integrate and fire, and exponential integrate and fire neuron meshes respectively. They can be called from  
514 the CLI. The scripts generate the three output files which any mesh generator script must produce: a *.mesh*  
515 file, a *.stat* file which defines extra cells in the mesh to hold probability mass that has settled at a stationary  
516 point, and a *.rev* file which defines a “reversal mapping” indicating how probability mass is transferred  
517 from strips in the mesh to the stationary cells. More information on *.mesh*, *.stat*, and *.rev* files is provided  
518 in the supplementary material section 6.

519 Once the *.mesh*, *.stat*, and *.rev* files have been generated by the user or by one of the automated 1D  
520 scripts, the Python command line interface, *miind.miindio*, provides commands to convert the three files  
521 into a single *.model* file and generate transition matrices stored in *.mat* files. The model file is what will be  
522 referenced and read by MIIND to load a mesh for a simulation. To generate this file, use the CLI command,  
523 **generate-model**. The command parameters are shown in Table 2. All input files must have the same base  
524 name, for example: *lif.mesh*, *lif.stat*, and *lif.rev*. If the command runs successfully, a new file will be created:  
525 *basename.model*. A number of pre-generated models are available in the *examples* directory of the MIIND  
526 repository to be used “out of the box” including the adaptive exponential integrate and fire and conductance  
527 based neuron models.

### Listing 7. Generate a Model in the CLI

528 > generate-model lif -60.0 -30.0

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

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

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

553 The second method translates the actual vertices of each cell according to the efficacy and calculates the  
554 exact overlapping area with other cells. The method by which this is achieved is the same as that used to  
555 generate the transition matrix of the grid algorithm, described in section 2.1. This method provides much  
556 higher accuracy than Monte Carlo but is one order of magnitude slower (it takes a similar amount of time  
557 to perform Monte Carlo with 100 points per cell). For some meshes, it is crucial to include very small  
558 transitions between cells to properly capture the dynamics which justifies the need for the slower method.  
559 It also benefits from requiring no additional user input in contrast to the Monte Carlo method.

560 In *miind.miindio*, the command **generate-matrix** can be used to automatically generate each *.mat*  
 561 file. In order to work, there must be a *basename.model* file in the working directory. The **generate-**  
 562 **matrix** command takes six parameters which are described in Table 3. Listing 8 shows an example of the  
 563 **generate-matrix** command. If successful, two files are generated: *basename.mat* and *basename.lost*.

**Listing 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.

564 > generate-matrix adex 0.1 10 0.0 5.0 false

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

**Listing 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.

573 > regenerate-reset adex 7.0

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

### 3.3 Jump Files

576 In some models, it is helpful to be able to set the efficacy as a function of the state. For example,  
 577 to approximate adaptive behaviour where the post synaptic efficacy lowers as the membrane potential  
 578 increases. Jump files have been used in MIIND to simulate the Tsodyks-Markram (Tsodyks and Markram,  
 579 1997) synapse model as described in De Kamps et al. (2019). In the model, one variable/dimension is  
 580 required to represent the membrane potential, *V*, of the post-synaptic neuron and the second to represent  
 581 the synaptic contribution, *G*. *G* and *V* are then used to derive the post-synaptic potential caused by an  
 582 incoming spike. Before generating the transition matrix, each cell can be assigned its own efficacy for  
 583 which the transitions will be calculated. During generation, Monte Carlo points will be translated according  
 584 to that value instead of a constant across the entire mesh. When calling the **generate-matrix** command, a  
 585 separate set of three parameters is required to use this feature. The base name of the model file, the number  
 586 of Monte Carlo points per cell, and a reference to a *.jump* file which stores the efficacy values for each cell  
 587 in the mesh.

**Listing 10.** Generate a transition matrix with a jump file in the CLI

588 > generate-matrix adex 10 adex.jump

589 As with the files required to build the mesh, the jump file must be user generated as the efficacy values may  
 590 be non-linear and involve one or both of the dimensions of the model. The format of a jump file is shown  
 591 in listing 11. The *<Efficacy>* element of the XML file gives an efficacy value for both dimensions of the  
 592 model and is how the resulting transition matrix will be referenced in the simulation. The *<Translations>*  
 593 element lists the efficacy in both dimensions for each cell in the mesh.

**Listing 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.

```

594 <Jump>
595 <Efficacy>0.0 0.1</Efficacy>
596 <Translations>
597 0,0 0.0 0.1
598 1,0 0.0 0.1
599 1,1 0.0 0.10012
600 1,2 0.0 0.10045
601 ...
602 </Translations>
603 </Jump>

```

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

## 4 WRITING THE XML FILE

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

### 4.1 Algorithms

613 An *<Algorithm>* in the XML code describes the simulation method for a population in the network. The  
 614 nodes of the network represent separate instances of these algorithm elements. Therefore, many nodes can  
 615 use the same algorithm. Each algorithm has different parameters or supporting files but as a minimum, all  
 616 algorithms must declare a type and a name. Each algorithm is also implicitly associated with a “weight  
 617 type”. All algorithms used in a single simulation must be compatible with the weight type as it describes the  
 618 way that populations interact. The *<WeightType>* element of the XML file can take the values, “double”,  
 619 “DelayedConnection”, or “CustomConnectionParameters”. Which value the weight type element takes  
 620 influences which algorithms are available in the simulation and how the connections between populations  
 621 will be defined. The following sections cover all Algorithm types currently supported in MIIND. Table 4  
 622 lists these algorithms and their compatible weight types.

#### 4.1.1 RateAlgorithm

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

**Listing 12.** A RateAlgorithm definition with a constant rate of 100Hz.

```

627 <Algorithm name="Cortical Background Algorithm" type="RateAlgorithm">
628   <rate>100.0</rate>
629 </Algorithm>

```

---

### 630 4.1.2 MeshAlgorithm and MeshAlgorithmCustom

631 In section 3.2, we saw how to generate *.model* and *.mat* files. These are required to simulate a population  
 632 using the mesh algorithm. Algorithm type=MeshAlgorithm tells MIIND to use this technique. The model  
 633 file is referenced as an attribute to the Algorithm definition. The *TimeStep* child element must match that  
 634 which was used to generate the mesh. This value is quoted in the model file. As many *MatrixFile* elements  
 635 can be declared as are required for the simulation, each with an associated .mat file reference.

#### **Listing 13.** A MeshAlgorithm definition with two matrix files.

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

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

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

### 649 4.1.3 GridAlgorithm and GridJumpAlgorithm

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

#### **Listing 14.** A GridAlgorithm definition using the AvgV (membrane potential) rate method.

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

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

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

**Listing 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)

```

668 <Algorithm type="GridJumpAlgorithm" name="ALG_ADEX" modelfile="adex.model" tau_refractive="0.0"
669   ↪ transformfile="adex_0_0_0_0_.tm" start_v="-65.0" start_w="0.0">
670 <TimeStep>0.0001</TimeStep>
671 </Algorithm>
672 ...
673 <Connection In="BG_NOISE" Out="ADEX_NODE" num_connections="1" efficacy="-0.05" delay="0.0" stationary=
674   ↪ -65.0"/>
```

#### 675 4.1.4 Additional Algorithms

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

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

**Listing 16.** A RateFunctor algorithm definition in which the firing rate linearly increases to 100Hz over 0.1 seconds and remains at 100Hz thereafter.

```

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

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

#### 691 4.2 Nodes

692 The *<Node>* block lists instances of the Algorithms defined above. Each node represents a single  
 693 population in the network. To create a node, the user must provide the name of one of the algorithms  
 694 defined in the algorithm block which will be instantiated. A name must also be given to uniquely identify  
 695 this node. The type describes the population as wholly inhibitory, excitatory, or neutral. The type dictates  
 696 the sign of the post synaptic efficacy caused by spikes from this population. Setting the type to neutral  
 697 allows the population to produce both excitatory and inhibitory (positive and negative) synaptic efficacies.  
 698 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  
 700 *INHIBITORY* respectively.

**Listing 17.** Three nodes defined in the Nodes section using the types *NEUTRAL*, *INHIBITORY*, and *EXCITATORY* respectively.

```

701 <Nodes>
702 ...
703 <Node algorithm="GRIDALG_FN" name="POP_1" type="NEUTRAL" />
704 <Node algorithm="ALG_ADEX" name="ADEX_NODE" type="INHIBITORY" />
705 <Node algorithm="RATEFUNC_BACKGROUND" name="BG_NOISE" type="EXCITATORY" />
```

706 ...  
 707 </Nodes>

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

### 710 4.3 Connections

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

**Listing 18.** A simple double WeightType Connection with a single rate multiplier.

717 <WeightType>double</WeightType>  
 718 <Connections>  
 719 ...  
 720 <Connection In="RATEFUNC\_BACKGROUND" Out="WC\_POP">0.1</Connection>  
 721 ...  
 722 </Connections>

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

**Listing 19.** A DelayedConnection with number of connections = 10, efficacy = 0.1, and delay of 1ms.

735 <WeightType>DelayedConnection</WeightType>  
 736 <Connections>  
 737 ...  
 738 <Connection In="RATEFUNC\_BACKGROUND" Out="BURSTER">10 0.1 0.001</Connection>  
 739 ...  
 740 </Connections>

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

**Listing 20.** A MeshAlgorithmCustom definition for use with WeightType=CustomConnectionParameters and a Connection using the num\_connections, efficacy, and delay attributes.

```

748 <WeightType>CustomConnectionParameters</WeightType>
749
750 <Algorithms>
751 ...
752 <Algorithm type="MeshAlgorithmCustom" name="ALG_ADEX" modelfile="adex.model" >
753   <TimeStep>0.001</TimeStep>
754   <MatrixFile>adex_0.05_0_0_0_.mat</MatrixFile>
755   <MatrixFile>adex_-0.05_0_0_0_.mat</MatrixFile>
756 </Algorithm>
757 ...
758 </Algorithms>
759
760
761 <Connections>
762 ...
763 <Connection In="ALG_ADEX" Out="RG_E" num_connections="1" efficacy="0.05" delay="0.0"/>
764 ...
765 </Connections>
```

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

#### 770 4.4 SimulationRunParameter

771 The *<SimulationRunParameter>* block contains parameter settings for the simulation as a whole. The  
 772 sub-elements listed in Table 5 are required for a full definition. Although most of the sub-elements are  
 773 self explanatory, *t\_step* has the limitation that it must match or be an integer multiple of all time steps  
 774 defined by any MeshAlgorithm and GridAlgorithm instances. *master\_steps* is used only for the GPGPU  
 775 implementation of MIIND (section 5). It allows the user to set the number of Euler iterations per time  
 776 step to solve the master equation. By default, the value is 10. However, to improve accuracy or to avoid  
 777 blow-up in the case where the time step is too large or the local dynamics are unstable, *master\_steps* should  
 778 be increased.

#### 779 4.5 Reporting

780 The *<Reporting>* block is used to describe how output is displayed and recorded from the simulation.  
 781 There are three ways to record output from the simulation: Density, Rate, and Display. The *<Rate>*  
 782 element takes the node *name* and *t\_interval* as attributes and creates a single file in the output directory.  
 783 *t\_interval* must be greater than or equal to the simulation time step. At each *t\_interval* of the simulation, the  
 784 output activity of the population is recorded on a new line of the generated file. Although the element is  
 785 called “Rate”, if average membrane potential has been chosen as the activity of this population, this is what  
 786 will be recorded here. *<Density>* is used to record the full probability density of the given population node.  
 787 As density is only relevant for the population density technique, it can only be recorded from nodes which  
 788 instantiate the mesh or grid algorithm types. The attributes are the node *name*, *t\_start*, *t\_end*, and *t\_interval*  
 789 which define the simulation times to start and end recording the density at the given interval. A file which  
 790 holds the probability mass values for each cell in the mesh or grid will be created in the output directory  
 791 for each *t\_interval* between *t\_start* and *t\_end*. Finally, the *<Display>* element can be used to observe the  
 792 evolution of the probability density function as the simulation is running. If a *Display* element is added in

793 the XML file for a specific node, when the simulation is run, a graphical window will open and display the  
 794 probability density for each time step. Again, display is only applicable to algorithms involving densities.  
 795 Enabling the display can significantly slow the simulation down. However, it is useful for debugging the  
 796 simulation and furthermore, each displayed frame is stored in the output directory so that a movie can be  
 797 made of the node's behaviour. How to generate this movie is discussed later in section 7.1.

**Listing 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.

```
798 <Reporting>
799 ...
800   <Density node="S" t_start="0.0" t_end="6.0" t_interval="0.01" />
801   <Density node="D" t_start="0.5" t_end="1.5" t_interval="0.001" />
802   <Display node="S" />
803   <Display node="D" />
804   <Rate node="S" t_interval="0.0001" />
805   <Rate node="D" t_interval="0.0001" />
806 ...
807 </Reporting>
```

## 808 4.6 Variables

809 The *<Simulation>* element can contain multiple *<Variable>* sub-elements each with a unique name  
 810 and value. Variables are provided for the convenience of the user and can replace any values in the XML  
 811 file. For example, a variable named *TIME\_END* can be defined to replace the value in the *t\_end* element of  
 812 the *SimulationRunParameter* block. When the simulation is run, the value of *t\_end* will be replaced with  
 813 the default value provided in the Variable definition. Using variables makes it easy to perform parameter  
 814 sweeps where the same simulation is run multiple times and only the variable's value is changed. How  
 815 parameter sweeps are performed is covered in the supplementary material section 8. All values in a MIIND  
 816 XML script can be set with a variable name. The type of the Variable is implicit and an error will be thrown  
 817 if, say, a non-numerical value is passed to the *tau\_refractive* attribute of a *MeshAlgorithm* object.

**Listing 22.** A Variable definition. *TIME\_END* has a default value of 18.0 and is used in the *t\_end* parameter definition.

```
818 <Variable Name='TIME_END'>18.0</Variable>
819 ...
820 <t_end>TIME_END</t_end>
```

## 5 MIIND ON THE GPU

821 The population density techniques of the mesh and grid algorithms rely on multiple applications of the  
 822 transition matrix which can be performed on each cell in parallel. This makes the algorithms prime  
 823 candidates for parallelisation on the graphics card. In the CPU versions, the probability mass is stored  
 824 in separate arrays, one for each population/node in the simulation. For the GPGPU version, these are  
 825 concatenated into one large probability mass vector so all cells in all populations can be processed in parallel.  
 826 From the user's perspective, switching between CPU and GPU implementations is trivial. In the XML file  
 827 for a simulation which uses *MeshAlgorithm* or *GridAlgorithm*, to switch to the vectorised GPU version, the  
 828 Algorithm types must be changed to *MeshAlgorithmGroup* and *GridAlgorithmGroup*. All other attributes  
 829 remain the same. Only *MeshAlgorithmGroup*, *GridAlgorithmGroup*, and *RateFunctor/RateAlgorithm*  
 830 types can be used for a vectorised simulation. When running a MIIND simulation containing a group

831 algorithm from a Python script, instead of importing *miind.miindsim*, *miind.miindsimv* should be used. The  
 832 Python module *miind.run* is agnostic to the use of group algorithms so can be used as shown previously.

**Listing 23.** A MeshAlgorithmGroup definition is identical to a MeshAlgorithm definition except for the type.

```
833 <Algorithm type="MeshAlgorithmGroup" name="ALG_ADEX" modelfile="adex.model" >
834   <TimeStep>0.001</TimeStep>
835   <MatrixFile>adex_0.05_0_0_.mat</MatrixFile>
836   <MatrixFile>adex_-0.05_0_0_.mat</MatrixFile>
837 </Algorithm>
838 <Algorithm type="GridAlgorithmGroup" name="OSC" modelfile="fn.model" tau_refractive="0.0" transformfile=
839   ↪ "fn_0_0_0_0_.tmat" start_v="-1.0" start_w="-0.3" ratemethod="AvgV">
840 <TimeStep>0.00001</TimeStep>
841 </Algorithm>
```

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

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

## 6 RUNNING A MIIND SIMULATION IN PYTHON

855 As demonstrated in the quick start guide, the command **python -m miind.run** takes a simulation XML file  
 856 as a parameter and runs the simulation. A similar script may be written by the user to give more control  
 857 over what happens during a simulation and how output activity is recorded and processed. It even allows  
 858 MIIND simulations to be integrated into other Python applications such as TVB (Sanz Leon et al., 2013) so  
 859 the population density technique can be used to solve the behaviour of nodes in a brain-scale network (see  
 860 section 9). To run a MIIND simulation in a Python script, the module *miind.miindsim* must be imported  
 861 (or *miind.miindsimv* if the simulation uses MeshAlgorithmGroup or GridAlgorithmGroup and therefore  
 862 requires CUDA support). Listing 24 shows an example script which uses the following available functions  
 863 to control the simulation.

**Listing 24.** A simple python script for running a MIIND simulation and plotting the results.

```
864 import matplotlib.pyplot as plt
865 import miind.miindsim as miind
866
867 miind.init(1, "lif.xml")
868
869 timestep = miind.getTimeStep()
870 simulation_length = miind.getSimulationLength()
871 print('Timestep from XML : {}'.format(timestep))
```

```

872 print('Sim time from XML : {}'.format(simulation_length))
873
874 miind.startSimulation()
875
876 constant_input = [2500]
877 activities = []
878 for i in range(int(simulation_length/timestep)):
879     activities.append(miind.evolveSingleStep(constant_input)[0])
880
881 miind.endSimulation()
882
883 plt.figure()
884 plt.plot(activities)
885 plt.title("Firing Rate.")
886
887 plt.show()

```

## 6.1 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 9). 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 25.** Calling *init* for a MIIND simulation lif.xml with the Variable SIM\_TIME set to 0.4.

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

## 6.2 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.

## 6.3 startSimulation()

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

## 6.4 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 6.6 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.

---

**917 6.5 endSimulation()**

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

**920 6.6 Additional XML Code for Python Support**

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

**Listing 26.** Special connection types for use in Python.

```
926 <Connections>
927 ...
928 <IncomingConnection Node="E">1 0.01 0</IncomingConnection>
929 <OutgoingConnection Node="E"/>
930 ...
931 </Connections>
```

932 Listing 26 defines an input to node E which will be interpreted as a DelayedConnection with the number  
933 of connections equal to 1 and a post synaptic efficacy of 0.01. No delay is defined here although it is  
934 permitted. OutgoingConnections are used to declare which nodes in the population network will pass their  
935 activity back to the Python script after each iteration. If the two connections in the listing are the only  
936 instances of IncomingConnection and OutgoingConnection, then the *evolveSingleStep* function will expect  
937 as a parameter, a list with one numeric value to represent the incoming rate to node E. *evolveSingleStep*  
938 will return a list with a single numeric value representing the activity of node E. In cases where there are  
939 more than one IncomingConnection, the order of values in the Python list parameter to *evolveSingleStep* is  
940 the same as the order of IncomingConnections defined in the XML. Similarly with OutgoingConnections,  
941 the order of the list of activities returned from *evolveSingleStep* is the same as the order of declaration in  
942 the XML file.

**7 USING THE CLI TO QUICKLY VIEW RESULTS**

943 Once a simulation has been run, either using *miind.run* or from a user written Python script, the  
944 *miind.miindio* CLI can be used to quickly plot the recorded results. As mentioned, the commands used  
945 in *miind* are based on the module *miind.miind\_api* and are reproducible in a Python script. However, it  
946 can be convenient to be able to run them directly from the command line to aid fast prototyping and bug  
947 fixing of models and simulations. The following section lists some common commands in the CLI and  
948 their usage. The accompanying files for this example are in the *examples/cli\_plots* directory. The following  
949 command starts the CLI and presents the user with a prompt:

**Listing 27.** Run the CLI.

```
950 $ python -m miind.miindio
```

951 When *miind.miindio* is called for the first time in a working directory, the user must identify the XML file  
952 which will describe the current working simulation. MIIND stores a reference to this file in a settings file in  
953 the working directory so that all subsequent commands will reference this simulation. Even if *miind.miindio*  
954 is quit and restarted, the current working simulation will be used as the context for commands until a new

955 current working simulation is defined or if it is called in a different directory. The user can set the current  
956 working simulation with the **sim** command.

**Listing 28.** Load a simulation file in the CLI.

957 > sim example.xml

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

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

**Listing 29.** Plot the rate of population POP1 in the CLI.

965 > rate POP1

966 Even while a simulation is running, calling **rate** in the CLI will plot the recorded activity up to the latest  
967 simulated time point. This is useful to keep an eye on the simulation as it progresses without waiting for  
968 completion. An example of the plots produced by **rate** is shown in Fig. 11A.

969 For populations using the grid or mesh algorithms, the user can call the **plot-density** command with  
970 parameters identifying the required node name and simulation time.

**Listing 30.** Plot the probability density of population POP1 at time 0.42s in the CLI.

971 > plot-density POP1 0.42

972 This command renders the mesh or grid and its population density at the given simulation time. When  
973 reading the simulation time parameter in the command, MIIND expects the time to be an integer multiple  
974 of the time step and to be expressed up to its least significant figure (for example, 0.1 instead of 0.10).  
975 Again, this command can be run during a simulation providing the time has been simulated. An example of  
976 a density plot is shown in Fig. 11B.

977 Similar to **plot-density**, **plot-marginals** can be used to display the marginal densities of a given population  
978 at a given time. Both marginals are plotted next to each other. The details of how marginal densities are  
979 calculated are explained in the supplementary material section 5. Fig. 11C shows an example of a marginal  
980 density plot.

**Listing 31.** Plot the marginal distributions of population POP1 at time 0.42s in the CLI.

981 > plot-marginals POP1 0.42

## 982 7.1 Generate a Density Movie

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

990 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  
 991 simulation. Finally, a name for the video file must be given.

**Listing 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.

992 > generate-density-movie POP1 512 0.1 pop1\_mov

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

**Listing 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.

997 > generate-marginal-movie POP1 512 0.1 pop1\_marginal\_mov

## 8 DESCRIPTION OF MIIND'S ARCHITECTURE AND FUNCTIONALITY

998 The main architectural concerns in MIIND relate to the two C++ libraries, MPILib and TwoDLib. MPILib  
 999 is responsible for instantiating and running the simulation. TwoDLib contains the CPU implementations of  
 1000 the grid and mesh algorithms. It is also responsible for generating transition matrices. Of the remaining  
 1001 libraries, GeomLib contains a population density technique implementation of neuron models with one  
 1002 time dependent variable, although it is also possible and indeed preferable to use the TwoDLib code for  
 1003 one dimensional models. EPFLLib and NumtoolsLib contain helper classes and type definitions. Fig. 12  
 1004 shows a reduced UML diagram of the MIIND C++ architecture. The aim of this section is to give a brief  
 1005 overview of the C++ MIIND code as a starting point for developers. The CUDA implementation of MIIND  
 1006 is similar in structure to the CPU solution and is available in the CudaTwoDLib and MiindLib libraries. A  
 1007 description of the differences is given in section 3 of the supplementary material.

### 8.1 MPILib

1009 The MPINetwork class in MPILib represents a simulation as a whole and is instantiated in the *init*  
 1010 function of the SimulationParserCPU class which is a specialisation of MiindTvbModelAbstract. *init*  
 1011 is called from the Python module and, as the name suggests, MiindTvbModelAbstract was originally  
 1012 written with the aim of Python integration into TVB. MPINetwork exposes member functions for building a  
 1013 network of nodes where each node is an instance of a neuron population which can be connected together so  
 1014 that the output activity from one population is input to another. The class also contains all of the simulation  
 1015 parameters such as the simulation length and time step. Finally, the MPINetwork class exposes a function  
 1016 to run the simulation in its entirety or take a single evolve step for use in an external control loop.

1017 Each node in the population network is represented by an instance of the MPINode class. A node  
 1018 has a name and an ID which is used to uniquely identify it in the simulation. A node also contains an  
 1019 implementation of AlgorithmInterface performing the integration technique required for this population  
 1020 (for example, GridAlgorithm or MeshAlgorithm). The *NodeType* describes whether a population should be  
 1021 thought of as excitatory or inhibitory. As discussed earlier, MIIND performs a validation check that the  
 1022 synaptic efficacy from a node is positive or negative respectively (or neutral). During each iteration, each  
 1023 node is responsible for consolidating the activity of all input connections, calling the integration step in the  
 1024 AlgorithmInterface implementation, and reporting the density and output activity (the average firing rate or  
 1025 membrane potential).

1026 In MPILib, a number of implementations of AlgorithmInterface are defined which can be instantiated  
1027 in a node. Implementations of AlgorithmInterface are responsible for the lion's share of the computation  
1028 in MIIND as this is where the integration of the model is performed. The interface is extremely simple,  
1029 providing a function to set parameters, an optional function for a preamble before each iteration, and  
1030 the *evolveNodeState* function to be called every time step. GridAlgorithm and MeshAlgorithm are imple-  
1031 ments of this interface defined in TwoDLib. MPILib and GeomLib hold the implementations of the  
1032 remaining algorithms available to the user which were discussed in section 4. Finally, the weight types,  
1033 DelayedConnection and CustomConnectionParameters are also defined in MPILib. All classes are C++  
1034 templates which take the weight type as a parameter to avoid code duplication and to enforce that only  
1035 algorithms with the same weight type can be used together.  
1036

## 8.2 TwoDLib

1037 As with the population models in MPILib and GeomLib, GridAlgorithm and MeshAlgorithm are  
1038 implementations of the AlgorithmInterface. We will focus here on the grid algorithm implementation  
1039 although the mesh algorithm uses the same structures or specialisations of those structures to perform  
1040 similar tasks as set out in section 3. GridAlgorithm is supported by two important classes. **Ode2DSys**  
1041 transfers probability mass according to the reset mapping of the *.model* file and calculates the average firing  
1042 rate of the population. In MeshAlgorithm, Ode2DSys also performs the pointer update for shifting  
1043 probability mass down the strips of the mesh. **MasterGrid** is responsible for solving the Poisson master  
1044 equation using a transition matrix calculated at simulation time based on the desired efficacy and grid  
1045 cell size. For each iteration, the function *evolveNodeState* is called which performs the main steps of the  
1046 population density algorithm.

1047 First, in GridAlgorithm, the deterministic dynamics are solved by applying the pre-generated transition  
1048 matrix once. The second step is a call to *Ode2DSys.RedistributeProbability()* to perform any reset  
1049 mappings for probability mass which appeared in the threshold cells last iteration. This step is useful for  
1050 neuron models, such as leaky integrate and fire, which contain an instruction to reset one or more variables  
1051 to a different value upon reaching a threshold.

1052 The third step calls on the MasterGrid class to solve the master equation for the incoming Poisson spike  
1053 rates from every incident node. MasterGrid begins with the current state of the probability mass distribution  
1054 across the grid, that is, the probability mass values of each cell in the grid. As described in section 2, every  
1055 cell has the same relative transition of probability mass due to a single incoming spike. For the whole  
1056 grid, this single transition is duplicated into a transition matrix which can be applied to the full probability  
1057 mass vector. Because there are at most two cells into which probability mass is transferred, this matrix  
1058 is extremely sparse and can be stored efficiently in a compressed sparse row (CSR) matrix. In the mesh  
1059 algorithm, this matrix is loaded from the *.mat* file.

1060 MeshAlgorithm requires a fourth step to transfer probability mass from the ends of strips to stationary  
1061 cells subject to a reversal mapping generated during the pre-processing phase. This is discussed in the  
1062 supplementary material section 6.

1063 Finally, SimulationParserCPU is an extension of the MiindTvbModelAbstract class used to parse the  
1064 simulation XML file and instantiate an MPINetwork object with the appropriate nodes and connections. Its  
1065 extensions of the functions declared in MiindTvbModelAbstract are exposed to the Python module to be  
1066 called from a Python script.

## 9 DISCUSSION

### 1067 MIIND fulfills a need for insight into neural behaviour at mesoscopic scales.

1068 The MIIND population density technique allows researchers to simulate population level behaviour by  
1069 defining the behaviour of the underlying neurons. This is in contrast to many rate based models which  
1070 describe the population behaviour directly. An example of how population behaviour can differ from the  
1071 underlying neuron model can be seen in the behaviour of a population of bursting neurons such as the  
1072 Izhikevich simple model. A single Izhikevich neuron with a constant input current or input spike rate  
1073 oscillates between a bursting period of repeated firing and a quiescent period of no firing. The average  
1074 behaviour of a population of Izhikevich neurons is different. Initially, all neurons are synchronised, they  
1075 burst and quiesce at the same time producing an oscillatory pattern of average firing rate in the population.  
1076 However, due to the random nature of Poisson input spikes, the neurons de-synchronise over time and the  
1077 average firing rate of the whole population damps to a constant value because only a subset of neurons  
1078 are bursting at any one time. Fig. 11A shows the damping of the output firing rate oscillations and the  
1079 ‘desynchronised’ density of a population of Izhikevich simple neurons.

### 1080 TVB Integration

1081 The Virtual Brain (Sanz Leon et al., 2013) and MIIND are both systems which facilitate the development  
1082 of neural mass or mean field population models with explicit descriptions of how multiple populations are  
1083 connected. Using these systems, the complex dynamics arising from the interaction of populations can be  
1084 studied. TVB provides a framework to describe a network of nodes (the connectivity) which, while it can be  
1085 abstract, generally represents regions of the human or primate brain. Connections between nodes represent  
1086 white matter tracts which transfer signals from one node to the next based on length and propagation speed.  
1087 TVB also allows the description of “coupling” functions which modulate these signals as they pass from  
1088 one node to another. Typically, the number of nodes is in the order of 100 or so. However, TVB also allows  
1089 for the definition of a “surface” which can be associated with 10s of thousands of nodes to simulate output  
1090 from common medical recording techniques such as EEG and BOLD fMRI. TVB has impressive clinical  
1091 relevance as well as supporting more theoretical neuroscience research. Users can build simulations using  
1092 the graphical user interface or directly using the Python source code.

1093 While MIIND and TVB have many functional similarities, both have differing strengths with respect to  
1094 the underlying simulation techniques and surrounding infrastructure. It was therefore clear that integrating  
1095 the smaller system, MIIND, into the more developed infrastructure of TVB might yield benefits from both.

1096 Although it is possible to model delayed connections and synaptic dynamics between populations in  
1097 MIIND, TVB provides a comprehensive method of defining such structures and behaviours through the  
1098 connectivity network and coupling functions. Some users of MIIND may find it useful and appropriate to  
1099 house their simulations in such a structure.

1100 TVB uses a number of model classes to describe the behaviour of the nodes in a network. When the  
1101 simulation is run, an instantiation of a specified model class takes the signals which have passed through  
1102 the network to arrive at each node and integrates forward by one time step (depending on the integration  
1103 method). In order to use MIIND nodes in TVB, a specialised model class was created to import the MIIND  
1104 Python library, instantiate it, then make a call to *evolveSingleStep()* in place of the integration function. The  
1105 inputs and outputs of *evolveSingleStep()* are treated by TVB as any other model. As the MIIND Python  
1106 library takes a simulation file name as a parameter to its *init* function, a single additional model class is all  
1107 that is required to expose any MIIND simulation to TVB. Fig. 13 shows the results from a simulation of  
1108 the TVB default whole-brain connectivity with populations of Izhikevich simple neurons in MIIND. The

1109 script and simulation files are available in the *examples/miind\_tvb* directory of the MIIND repository. Both  
1110 **Reasoning about probability density instead of populations of individual neurons**  
1111 **simplifies output analysis.**

1113 The output firing rate or membrane potential of a MIIND population which uses the the mesh algorithm  
1114 or grid algorithm is devoid of any variation which you would see from a population of individual neurons.  
1115 This is because the effect of Poisson generated input spike trains is applied to a probability density function,  
1116 effectively an infinite population of neurons. Spike train inputs to a finite population of neurons produces  
1117 variation in how individual neurons move through state space resulting in noisy output rates at the population  
1118 level. While this can be mitigated using a larger number of neurons, the use of smoothing techniques, or  
1119 curve fitting, MIIND requires none of these methods to produce an output which is immediately clear to  
1120 interpret. For example, MIIND was used to build and simulate a spinal circuit model using populations  
1121 of integrate and fire neurons (York et al., 2019). The average firing rates of the populations were used to  
1122 compare patterns of activity with results from an EMG experiment. As the patterns to be observed were  
1123 on the order of seconds, there was no need to capture faster variation in activity from the simulation and  
1124 indeed, a direct simulation would have produced output which may have obscured these patterns.

1125 MIIND has also been used to simulate central pattern generator models which rely on mutually inhibiting  
1126 populations of bursting neurons. The interaction of the two populations significantly influences their sub-  
1127 threshold dynamics. In particular, it can be difficult to identify the dynamics responsible for the swapping  
1128 of states from bursting to quiescent (escape or release). Observing the changing probability density function  
1129 **Handling Noise** makes it very clear how the two populations are behaving.

1131 A major benefit of MIIND's population density technique is the ability to observe the effect of noise on a  
1132 population, and to manipulate noise in an intuitive way. For a given simulation, the Poisson distributed  
1133 input to a population causes a spread of probability mass across the state space as some neurons receive  
1134 many spikes, and some receive fewer. It is explained in de Kamps (2013) how the Poisson input causes a  
1135 mean increase in membrane potential equal to the product of the post synaptic efficacy,  $h$ , and the average  
1136 input rate,  $\nu$ . It causes a variance equal to  $\nu h^2$ .  $h$  and  $\nu$  can therefore be set such that the mean remains the  
1137 same but the variance changes to observe the effect of noise on the population.

1138 Another simple way to increase the variance of the population is to introduce two additional inputs with  
1139 equal rates and opposite post-synaptic efficacies. Again, the mean increase caused by the input remains  
1140 unchanged but the variance can be increased significantly and this requires only a small change to the XML  
1141 simulation file.

#### 1142 **A model agnostic system at the population level makes prototyping quick and intuitive.**

1144 Because MIIND provides insight of how a neuron model produces behaviour at the population level, it is  
1145 beneficial that the grid algorithm enables the user to quickly reproduce the *.model* and *.tmat* files if the  
1146 underlying neuron model needs to be changed. An example of this can be observed in a half-centre oscillator  
1147 made of a pair of mutually inhibiting populations of bursting neurons. The frequency of oscillation can be  
1148 made dependent or independent of the input spike rate by including a limit on the slow excitability variable  
1149 of the underlying neuron model. To make this change, the user can alter the neuron model then rebuild the  
1150 *.model* and *.tmat* file and no change to the population level network is required.

1151

**1152 DiPDE**

1153 DiPDE (DiPDE, 2015; Iyer et al., 2013) is an alternative implementation of the population density  
1154 technique for one dimensional neuron models. It does not employ the “mesh” discretisation method used  
1155 in the MIIND mesh algorithm and has primarily been used with populations of leaky integrate and fire  
1156 neurons. DiPDE can be used to simulate the Potjans-Diesmann microcircuit model (Cain et al., 2016)  
1157 which shows good agreement with MIIND (Fig. 5). MIIND is a much larger application than DiPDE  
1158 because it allows users to design their own underlying neuron models for each population using either the  
1159 mesh or grid algorithms.

1160

**1161 Future Work**

1162 A limitation on the MIIND population density technique is that a maximum of two time-dependent  
1163 variables can be used to describe the underlying neuron model of each population. In the mesh algorithm,  
1164 for higher dimensions, mesh building would need to be automated but this is not a trivial problem to solve.  
1165 The grid algorithm, however, is entirely automated and work has been done to extend MIIND for 3D  
1166 neuron models. Fig. 14 shows the 3D density plot of a population of Hindmarsh-Rose neurons in MIIND.  
1167 The technique used to generate the 2D transition matrices outlined in section 2 extends to N dimensions so  
1168 there is theoretically no limit to the dimensionality of the underlying neuron model in the grid algorithm.  
1169 However, both the grid algorithm and mesh algorithm suffer from “the curse of dimensionality” such that  
1170 with each additional variable, the number of cells to cover the state space increases to the point where the  
1171 memory and processing requirements are too high. Luckily, a great number of neuron behaviours can be  
1172 captured with only two or three time-dependent variables with appropriate approximations.

1173 Large networks can be built up quickly in MIIND. To add a node to a simulation file requires just a single  
1174 line. Integrating the node into the rest of the network with requisite connections is equally convenient. As  
1175 mentioned, the Potjans-Diesmann model has been implemented as a single cortical column but this is by  
1176 no means the limit of the size of network which can be built. It is feasible that a patch of cortex made of  
1177 perhaps hundreds of cortical columns can be simulated efficiently in MIIND. The benefit of such a network  
1178 would be to demonstrate how cortical columns interact together under different connectivity regimes and  
1179 inputs as well as providing the ability to quickly and easily “swap out” the underlying neuron model of  
1180 each population. Typically, LIF is used but adaptive integrate and fire would be a closer approximation to  
1181 pyramidal neurons in cortex.

**1182 Conclusion**

1183 We have presented the mesh and grid algorithms, MIIND’s population density techniques for simulating  
1184 populations of neurons, and given a full account of the software features available to users. While the  
1185 mesh algorithm was developed some time ago, the grid algorithm which was added to MIIND recently has  
1186 precipitated a more accessible, user friendly software package. We hope that the explanations given here  
1187 along with a lower technical barrier to entry will encourage researchers to make use of the tool.

**CONFLICT OF INTEREST STATEMENT**

1188 The authors declare that the research was conducted in the absence of any commercial or financial  
1189 relationships that could be construed as a potential conflict of interest.

**AUTHOR CONTRIBUTIONS**

1190 HO and MdK contributed to the text of this article. YML, MEL, DS, and LD contributed to the development  
1191 of the population density technique and MIIND software.

## FUNDING

1192 This project received funding from the European Union’s Horizon 2020 research and innovation programme  
1193 under Grant Agreement No. 720270 (HBP SGA1) and Specific Grant Agreement No. 785907 (Human  
1194 Brain Project SGA2) (MdK; YML). HO is funded by EPSRC. The funders had no role in study design,  
1195 data collection and analysis, decision to publish, or preparation of the manuscript.

## ACKNOWLEDGMENTS

1196 The authors wish to thank Frank van der Velde and Martin Perez-Guevara for their continued support of  
1197 the MIIND project.

## DATA AVAILABILITY STATEMENT

1198 The MIIND source code and installation packages are available as a github repository at <https://github.com/dekamps/miind>.

1200 MIIND can be installed for use in Python using “pip install miind” on many Linux, MacOS, and Windows  
1201 machines with python versions  $\geq 3.6$ .

1202 Documentation is available at <https://miind.readthedocs.io/>.

## REFERENCES

- 1203 Amit, D. J. and Brunel, N. (1997). Model of global spontaneous activity and local structured activity  
1204 during delay periods in the cerebral cortex. *Cerebral cortex (New York, NY: 1991)* 7, 237–252
- 1205 Bower, J. M. and Beeman, D. (2012). *The book of GENESIS: exploring realistic neural models with the*  
1206 *GENeral NEural SImulation System* (Springer Science & Business Media)
- 1207 Brette, R. and Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description  
1208 of neuronal activity. *Journal of neurophysiology* 94, 3637–3642
- 1209 Brunel, N. and Hakim, V. (1999). Fast global oscillations in networks of integrate-and-fire neurons with  
1210 low firing rates. *Neural computation* 11, 1621–1671
- 1211 Cain, N., Iyer, R., Koch, C., and Mihalas, S. (2016). The computational properties of a simplified cortical  
1212 column model. *PLoS computational biology* 12, e1005045
- 1213 Carlu, M., Chehab, O., Dalla Porta, L., Depannemaeker, D., Héricé, C., Jedynak, M., et al. (2020). A  
1214 mean-field approach to the dynamics of networks of complex neurons, from nonlinear integrate-and-fire  
1215 to hodgkin-huxley models. *Journal of neurophysiology* 123, 1042–1051
- 1216 D’Angelo, E., Antonietti, A., Casali, S., Casellato, C., Garrido, J. A., Luque, N. R., et al. (2016). Modeling  
1217 the cerebellar microcircuit: new strategies for a long-standing issue. *Frontiers in cellular neuroscience*  
1218 10, 176
- 1219 de Kamps, M. (2013). A generic approach to solving jump diffusion equations with applications to neural  
1220 populations. *arXiv preprint arXiv:1309.1654*
- 1221 De Kamps, M., Baier, V., Drever, J., Dietz, M., Mösenlechner, L., and Van Der Velde, F. (2008). The state  
1222 of miind. *Neural Networks* 21, 1164–1181
- 1223 De Kamps, M., Lepperød, M., and Lai, Y. M. (2019). Computational geometry for modeling neural  
1224 populations: From visualization to simulation. *PLoS computational biology* 15, e1006729
- 1225 DiPDE (2015). Website: © 2015 Allen Institute for Brain Science. DiPDE Simulator [Internet]. Available  
1226 from: <https://github.com/AllenInstitute/dipde>.
- 1227 El Boustani, S. and Destexhe, A. (2009). A master equation formalism for macroscopic modeling of  
1228 asynchronous irregular activity states. *Neural computation* 21, 46–100

- 1229 FitzHugh, R. (1961). Impulses and physiological states in theoretical models of nerve membrane.  
1230 *Biophysical journal* 1, 445
- 1231 Fourcaud-Trocmé, N., Hansel, D., Van Vreeswijk, C., and Brunel, N. (2003). How spike generation  
1232 mechanisms determine the neuronal response to fluctuating inputs. *Journal of Neuroscience* 23, 11628–  
1233 11640
- 1234 Furber, S., Galluppi, F., Temple, S., and Plana, L. (2014). The spinnaker project. *IEEE. Proceedings* 102,  
1235 652–665. doi:10.1109/JPROC.2014.2304638
- 1236 Gerstner, W. (1998). *Spiking neurons*. Tech. rep., MIT-press
- 1237 Gewaltig, M.-O. and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia* 2, 1430
- 1238 Hindmarsh, J. L. and Rose, R. (1984). A model of neuronal bursting using three coupled first order  
1239 differential equations. *Proceedings of the Royal society of London. Series B. Biological sciences* 221,  
1240 87–102
- 1241 Hines, M. L. and Carnevale, N. T. (2001). Neuron: a tool for neuroscientists. *The neuroscientist* 7, 123–135
- 1242 Iyer, R., Menon, V., Buice, M., Koch, C., and Mihalas, S. (2013). The influence of synaptic weight  
1243 distribution on neuronal population dynamics. *PLoS Comput Biol* 9, e1003248
- 1244 Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on neural networks* 14,  
1245 1569–1572
- 1246 Izhikevich, E. M. (2007). *Dynamical systems in neuroscience* (MIT press)
- 1247 Jirsa, V. K., Stacey, W. C., Quilichini, P. P., Ivanov, A. I., and Bernard, C. (2014). On the nature of seizure  
1248 dynamics. *Brain* 137, 2210–2230
- 1249 Kamps, M. d. (2003). A simple and stable numerical solution for the population density equation. *Neural  
1250 computation* 15, 2129–2146
- 1251 Knight, B. W. (1972). Dynamics of encoding in a population of neurons. *The Journal of general physiology*  
1252 59, 734–766
- 1253 Knight, B. W., Manin, D., and Sirovich, L. (1996). Dynamical models of interacting neuron populations in  
1254 visual cortex. *Robot Cybern* 54, 4–8
- 1255 Lai, Y. M. and de Kamps, M. (2017). Population density equations for stochastic processes with memory  
1256 kernels. *Physical Review E* 95, 062125
- 1257 Mattia, M. and Del Giudice, P. (2002). Population dynamics of interacting spiking neurons. *Physical  
1258 Review E* 66, 051917
- 1259 Mattia, M. and Del Giudice, P. (2004). Finite-size dynamics of inhibitory and excitatory interacting spiking  
1260 neurons. *Physical Review E* 70, 052903
- 1261 Montbrió, E., Pazó, D., and Roxin, A. (2015). Macroscopic description for networks of spiking neurons.  
1262 *Physical Review X* 5, 021028
- 1263 Nagumo, J., Arimoto, S., and Yoshizawa, S. (1962). An active pulse transmission line simulating nerve  
1264 axon. *Proceedings of the IRE* 50, 2061–2070
- 1265 Nykamp, D. Q. and Tranchina, D. (2000). A population density approach that facilitates large-scale  
1266 modeling of neural networks: Analysis and an application to orientation tuning. *Journal of computational  
1267 neuroscience* 8, 19–50
- 1268 Omurtag, A., Knight, B. W., and Sirovich, L. (2000). On the simulation of large populations of neurons.  
1269 *Journal of computational neuroscience* 8, 51–63
- 1270 Osborne, H. and De Kamps, M. (2021). *MIIND Documentation [Internet]*. Available from:  
1271 <https://miind.readthedocs.io>.
- 1272 Potjans, T. C. and Diesmann, M. (2014). The cell-type specific cortical microcircuit: relating structure and  
1273 activity in a full-scale spiking network model. *Cerebral cortex* 24, 785–806

- 1274 Proix, T., Bartolomei, F., Guye, M., and Jirsa, V. K. (2017). Individual brain structure and modelling  
1275 predict seizure propagation. *Brain* 140, 641–654
- 1276 Sanz Leon, P., Knock, S. A., Woodman, M. M., Domide, L., Mersmann, J., McIntosh, A. R., et al. (2013).  
1277 The virtual brain: a simulator of primate brain network dynamics. *Frontiers in neuroinformatics* 7, 10
- 1278 Traub, R. D., Contreras, D., Cunningham, M. O., Murray, H., LeBeau, F. E., Roopun, A., et al. (2005).  
1279 Single-column thalamocortical network model exhibiting gamma oscillations, sleep spindles, and  
1280 epileptogenic bursts. *Journal of neurophysiology* 93, 2194–2232
- 1281 Tsodyks, M. V. and Markram, H. (1997). The neural code between neocortical pyramidal neurons depends  
1282 on neurotransmitter release probability. *Proceedings of the national academy of sciences* 94, 719–723
- 1283 Uhlenbeck, G. E. and Ornstein, L. S. (1930). On the theory of the brownian motion. *Physical review* 36,  
1284 823
- 1285 Wilson, H. R. and Cowan, J. D. (1972). Excitatory and inhibitory interactions in localized populations of  
1286 model neurons. *Biophysical journal* 12, 1–24
- 1287 Wilson, M. A., Bhalla, U. S., Uhley, J. D., and Bower, J. M. (1988). Genesis: a system for simulating  
1288 neural networks
- 1289 York, G., Osborne, H., Sriya, P., Astill, S., De Kamps, M., and Chakrabarty, S. (2019). Muscles recruited  
1290 during an isometric knee extension task is defined by proprioceptive feedback. *BioRxiv* , 802736

## TABLES

**Table 1.** Parameters for the `grid-generate.generate` function.

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

**Table 2.** Parameters for the `generate-model` command in the CLI.

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

**Table 3.** Parameters for the `generate-matrix` command in the CLI.

Parameter Name	Notes
<i>basename</i>	The shared name of the <code>.model</code> , <code>.fid</code> (if required), and generated <code>.mat</code> files.
<i>v_efficiency</i>	The efficacy value in the <i>v</i> (membrane potential) direction. If the parameter <i>h_efficiency</i> is used, this should be zero.
<i>points / precision</i>	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 <code>.mat</code> file to the nearest $\frac{1}{precision}$ .
<i>h_efficiency</i>	The efficacy value in the <i>h</i> direction. If the parameter <i>v_efficiency</i> is used, this should be zero.
<i>reset-shift</i>	The shift in the <i>h</i> direction which neurons take when being reset.
<i>use_geometric</i>	A boolean flag set to “true” if the geometric method is used and “false” for Monte Carlo.

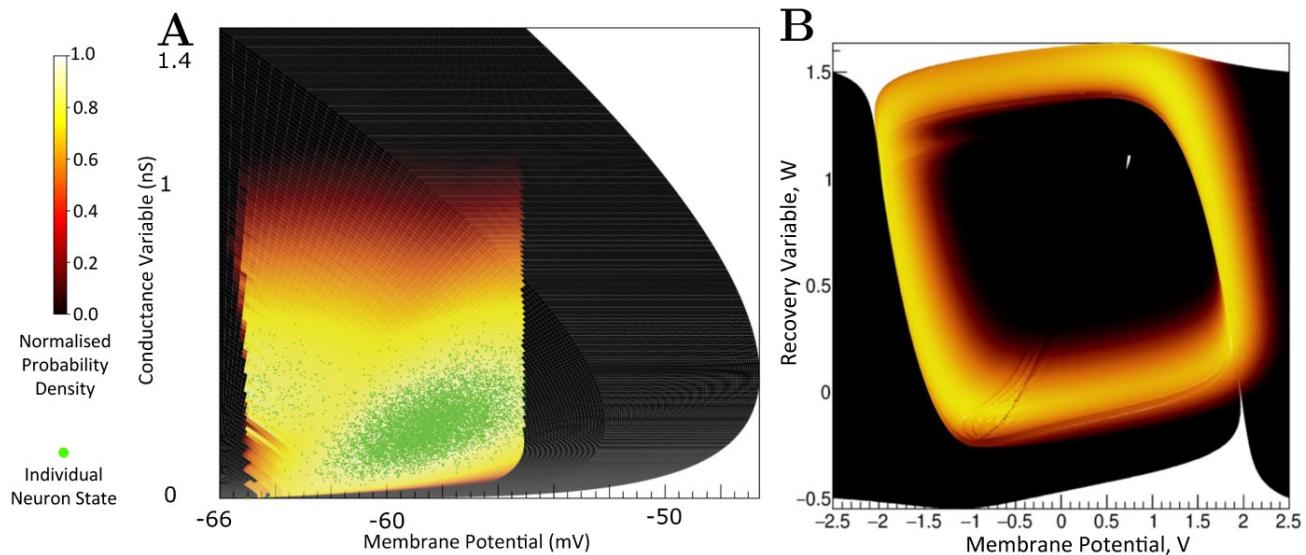
**Table 4.** Compatible weight types for each algorithm type defined in the simulation XML file.

Algorithm Name	double	DelayedConnection	CustomConnectionParameters
<i>RateAlgorithm</i>	✓	✓	✓
<i>MeshAlgorithm</i>		✓	
<i>MeshAlgorithmCustom</i>			✓
<i>GridAlgorithm</i>			✓
<i>GridJumpAlgorithm</i>			✓
<i>OUAlgorithm</i>		✓	
<i>WilsonCowanAlgorithm</i>	✓		
<i>RateFunctor</i>	✓	✓	✓

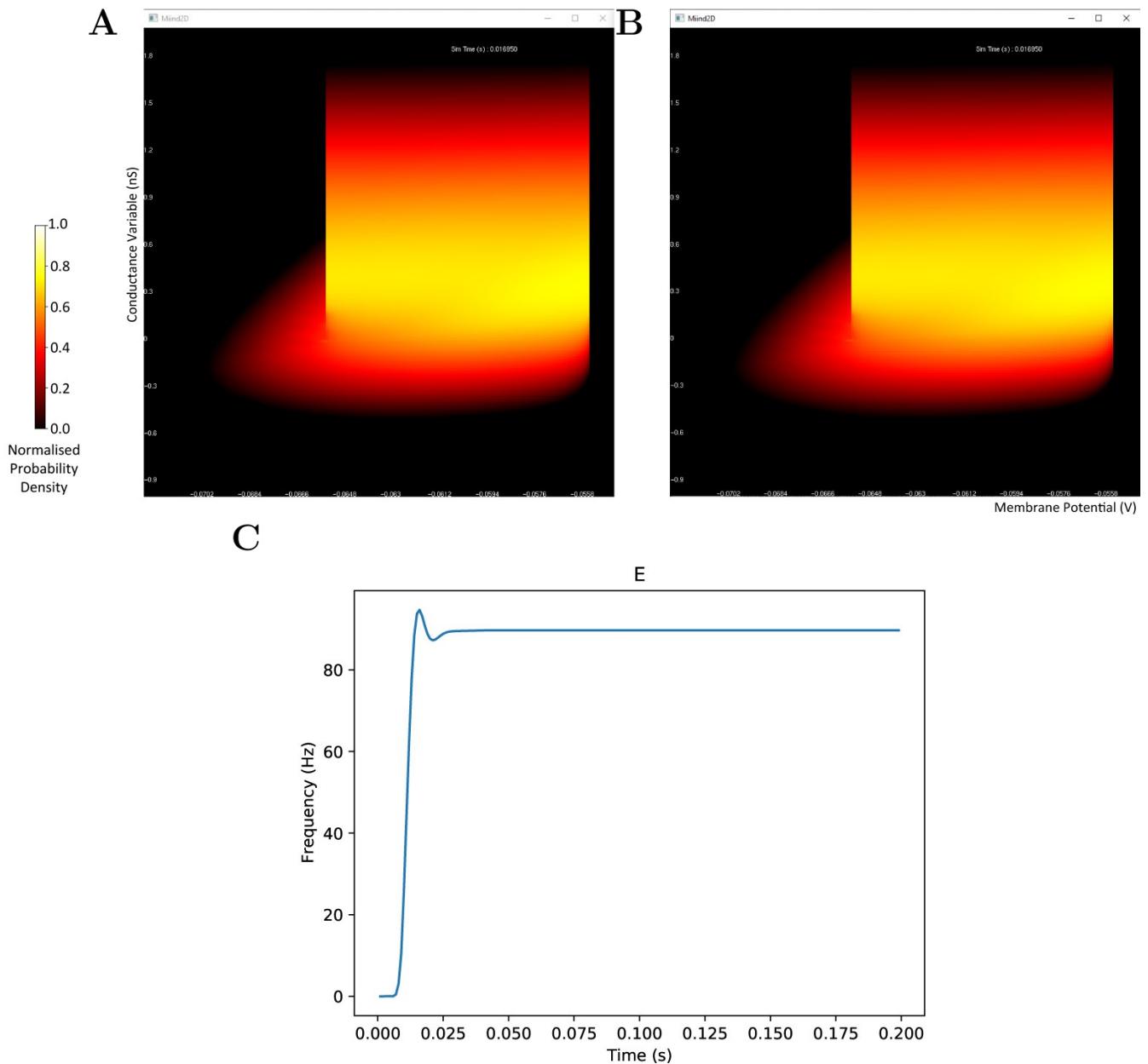
**Table 5.** The required sub-elements for the `SimulationRunParameter` section of the XML simulation file.

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

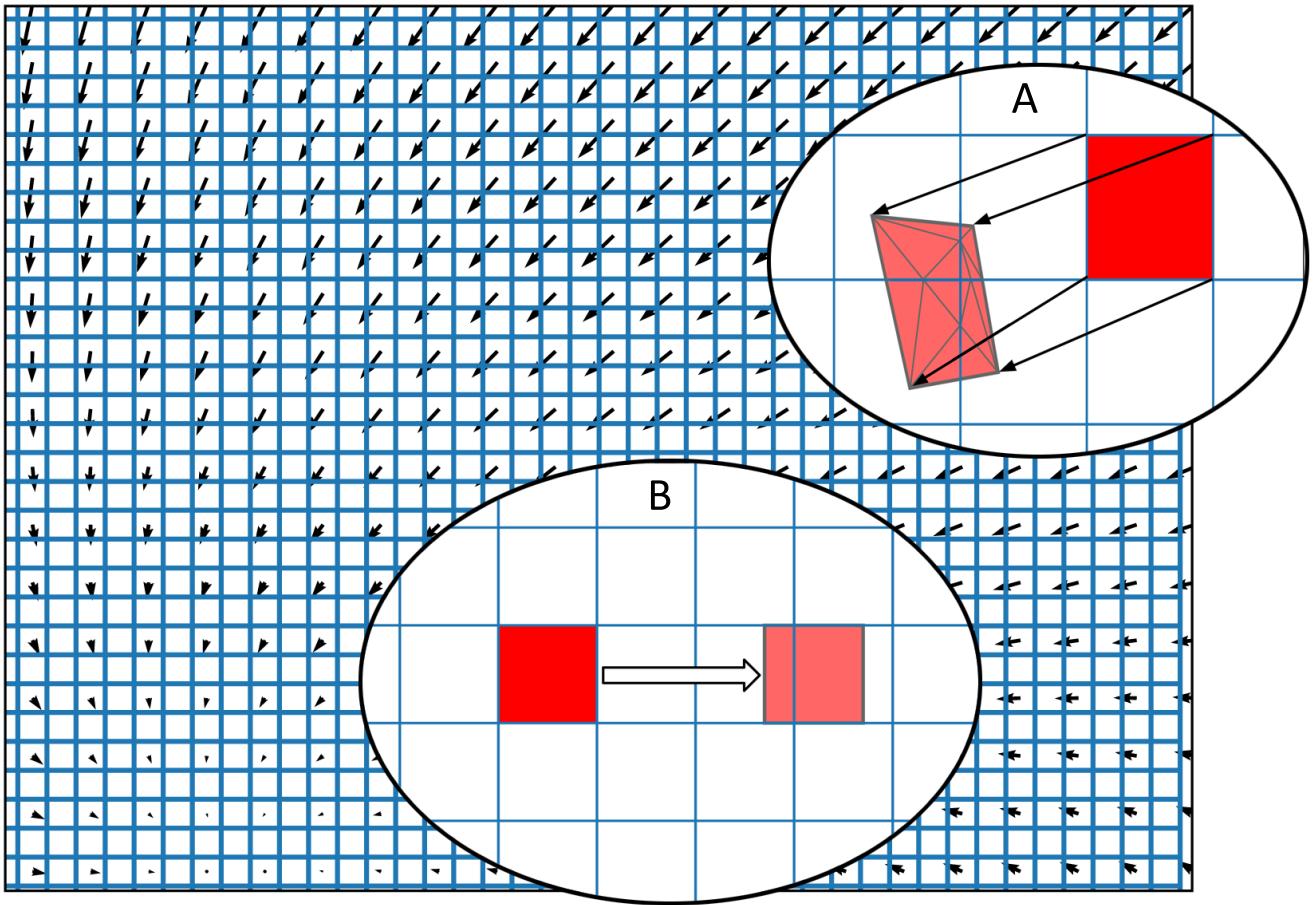
## FIGURES



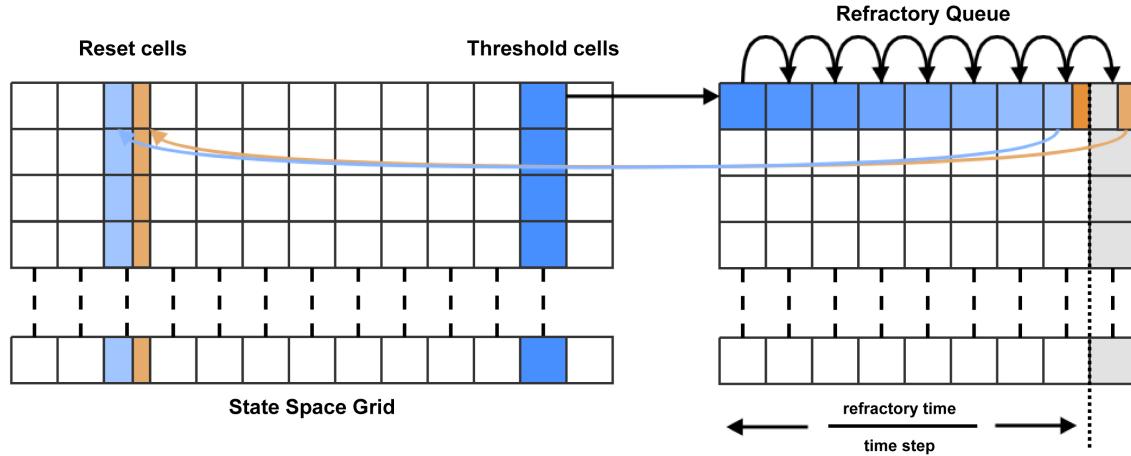
**Figure 1.** (A) The state space of a conductance based point model neuron. It is spanned by two variables: the membrane potential and a variable representing how open the channel is. This channel has an equilibrium potential that is positive. The green dots represent the state of individual neurons 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.



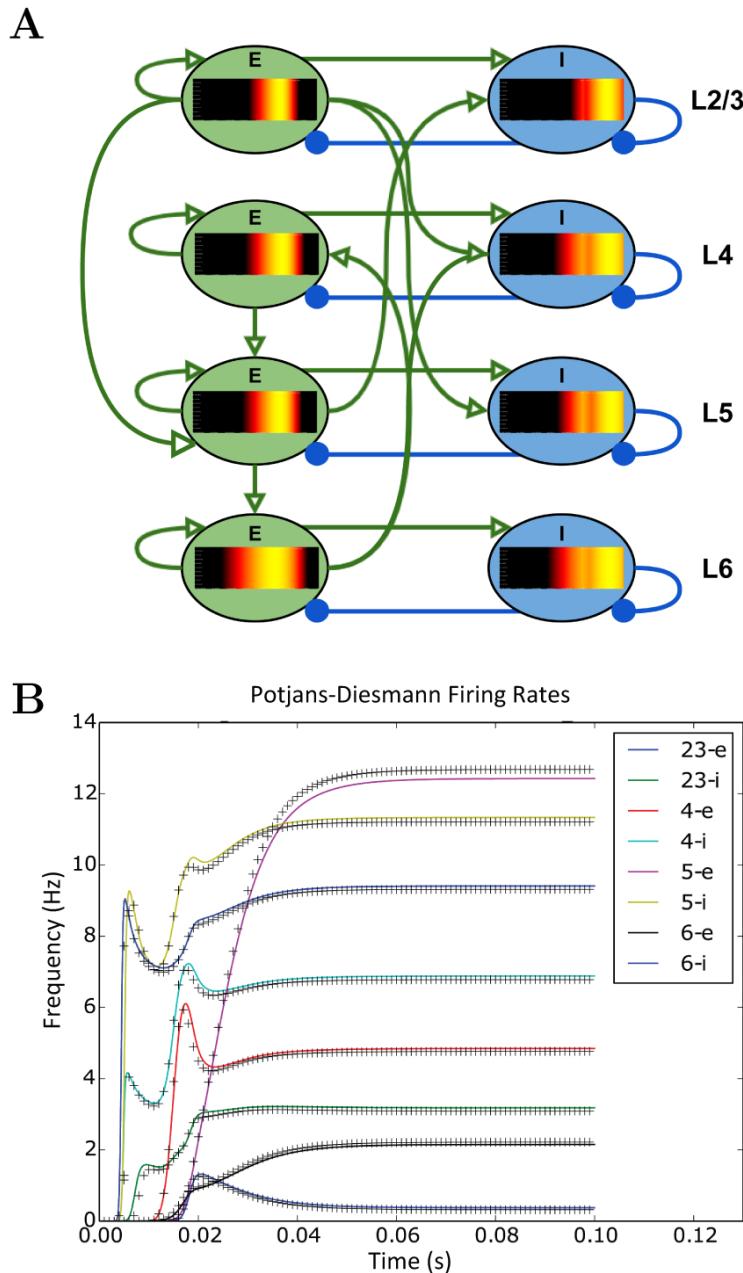
**Figure 2.** The display output of a running E-I population network simulation of conductance based neurons. (A) The probability density heat map (normalised to the maximum density value) of the excitatory population. (B) The probability density heat map of the inhibitory population. Brighter colours 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.



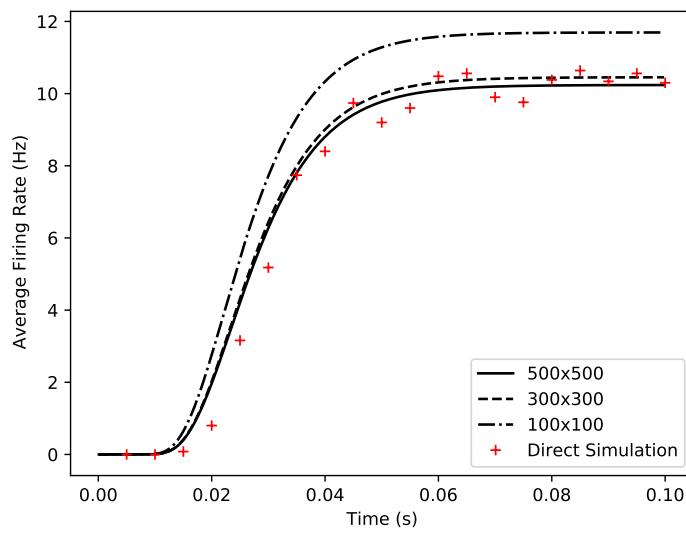
**Figure 3.** The state space of a neuron model (shown here as a vector field) is discretised into a regular grid of cells. (A) The transition matrix for solving the deterministic dynamics of the population is generated by applying a single time step of the underlying neuron model to each vertex of each 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.



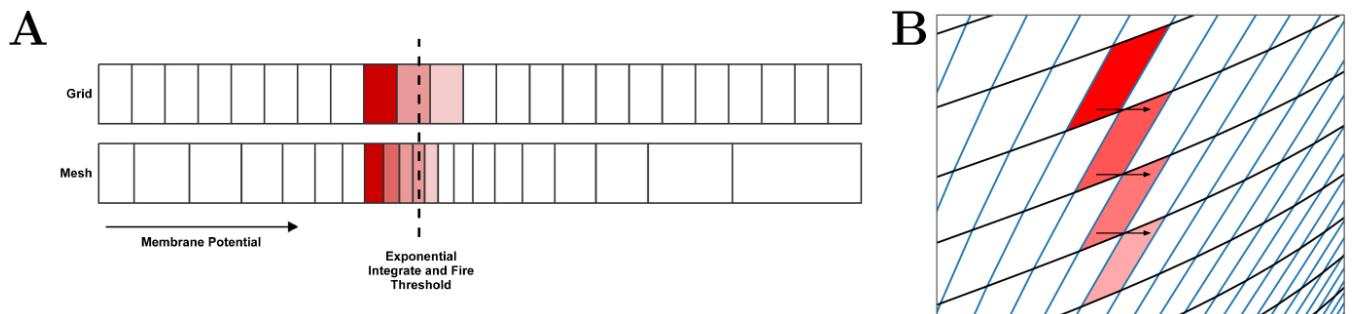
**Figure 4.** For each time step, probability mass in the cells which lie across the threshold (threshold cells) is pushed onto the beginning of the refractory queue. There is one queue per threshold cell. During each subsequent time step, the probability mass is shifted one place along the queue until 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.



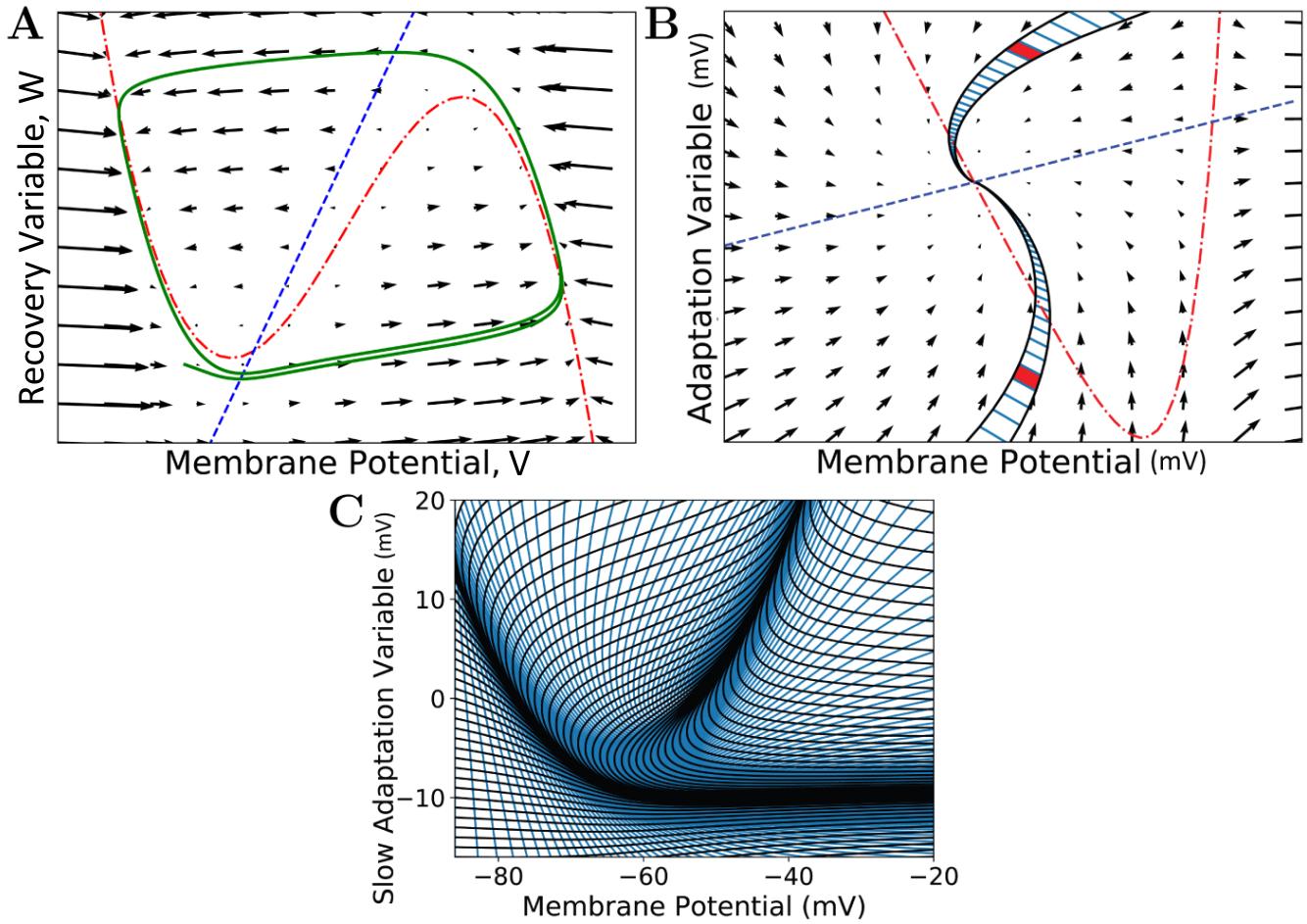
**Figure 5.** (A) A representation of the connectivity between populations in the Potjans-Diesmann microcircuit model. Each population shows the probability density at an early point in the simulation before all populations have reached a steady state. All populations are of leaky-integrate-and-fire 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).



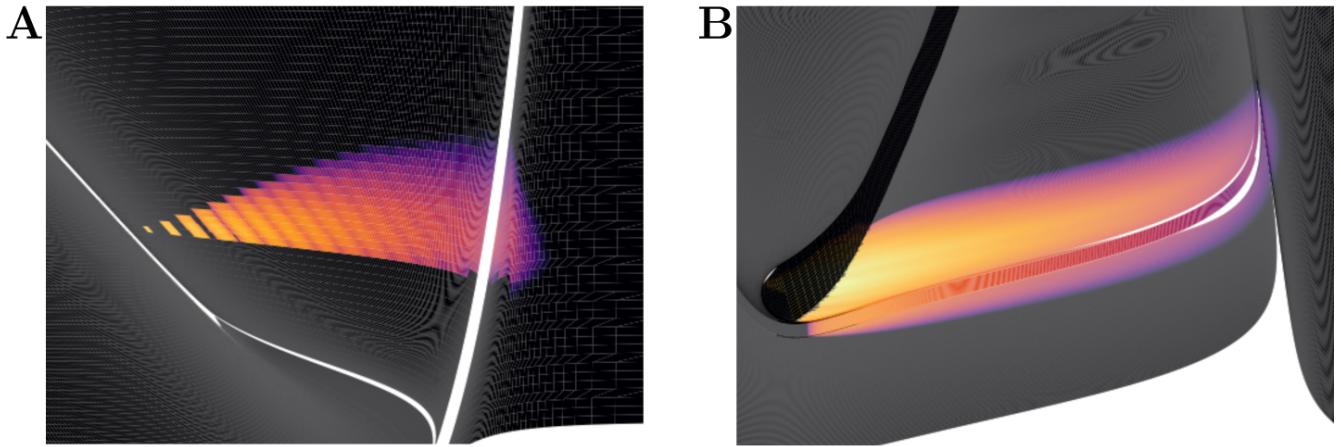
**Figure 6.** Comparison of average firing rates from four simulations of a single population of conductance based neurons. The black solid and dashed lines indicate MIIND simulations using the grid algorithm with different grid resolutions. The red crosses show the average firing rate of a direct simulation of 10,000 neurons.



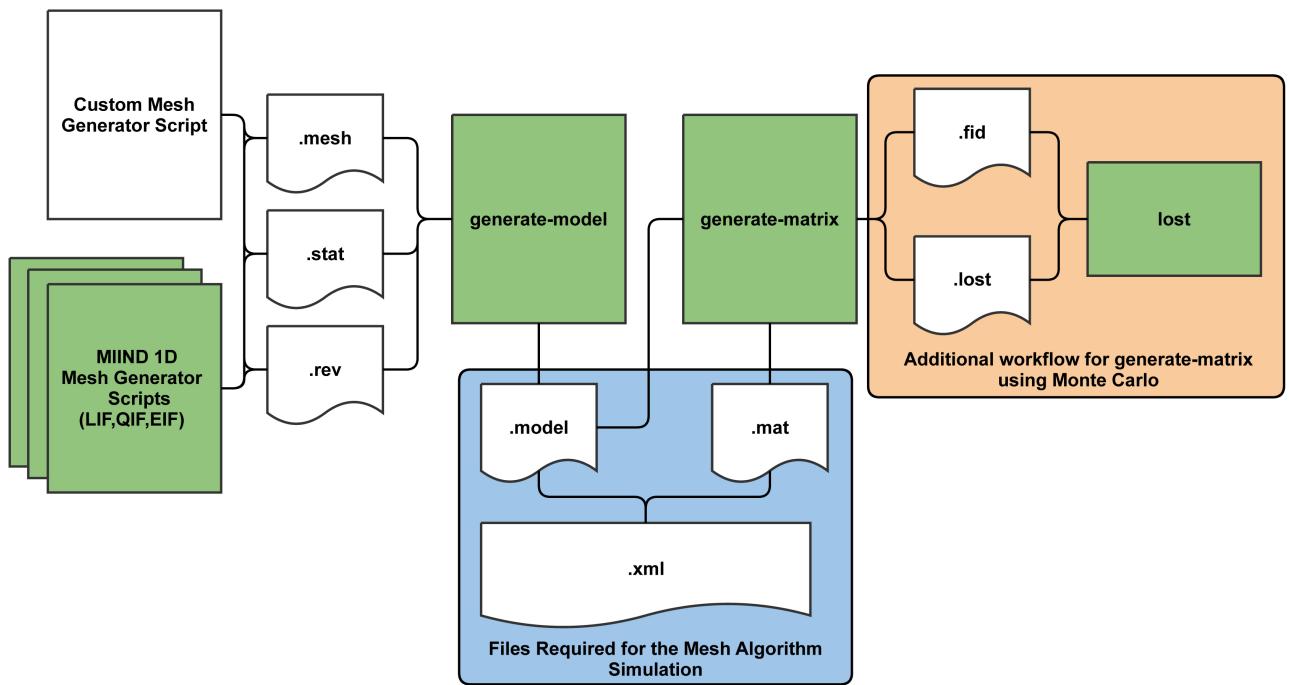
**Figure 7.** (A) In the grid algorithm, large cells cause probability mass to be distributed further than it should. This error is expressed most clearly in models where the average firing rate of the population is highly dependent on the amount of probability mass passing through an area of slow dynamics. (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.



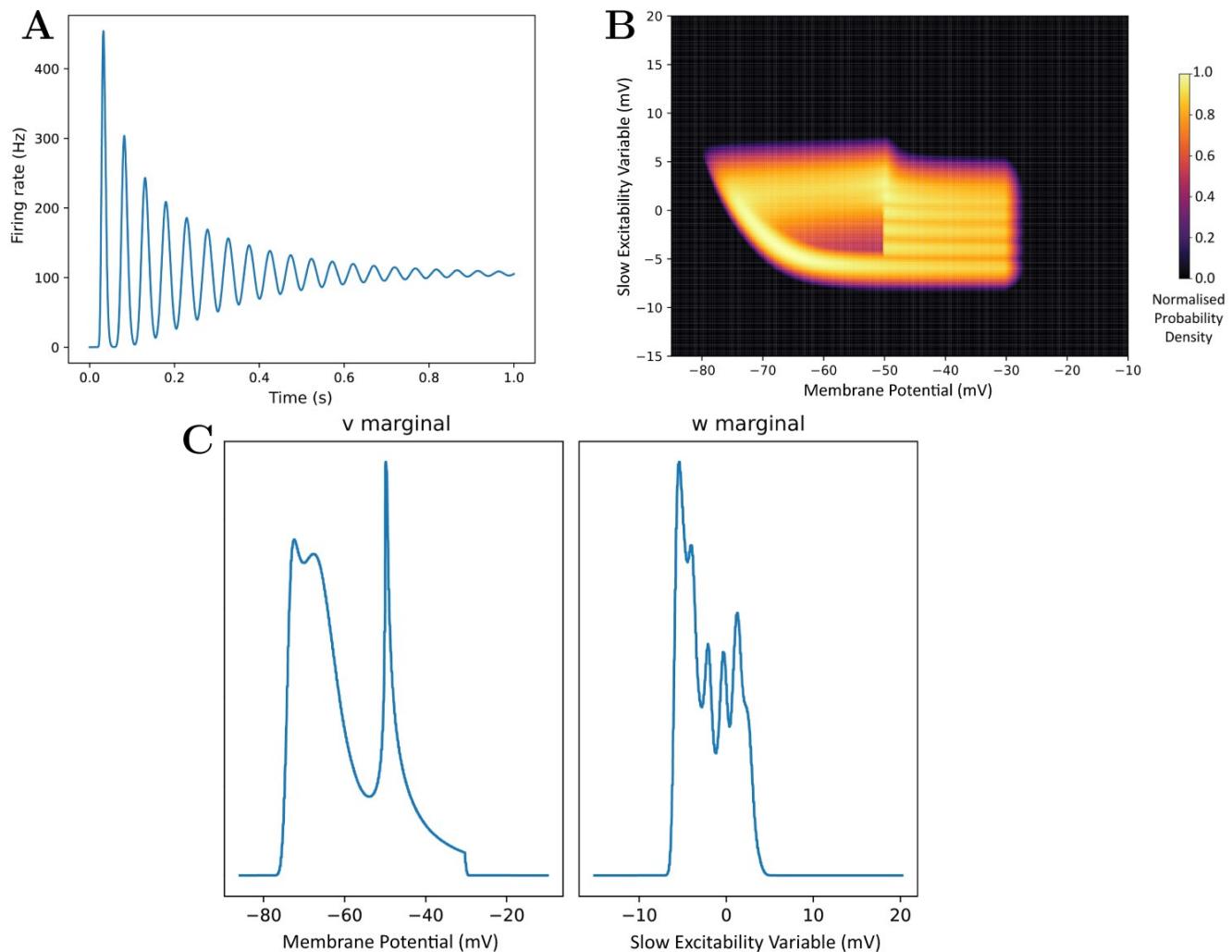
**Figure 8.** (A) A vector field of the FitzHugh-Nagumo neuron model (FitzHugh, 1961). Arrows show the direction of motion of states through the field according to the dynamics of the model. The red broken dashed nullcline indicates where the change in  $V$  is zero. The blue dashed nullcline indicates 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.



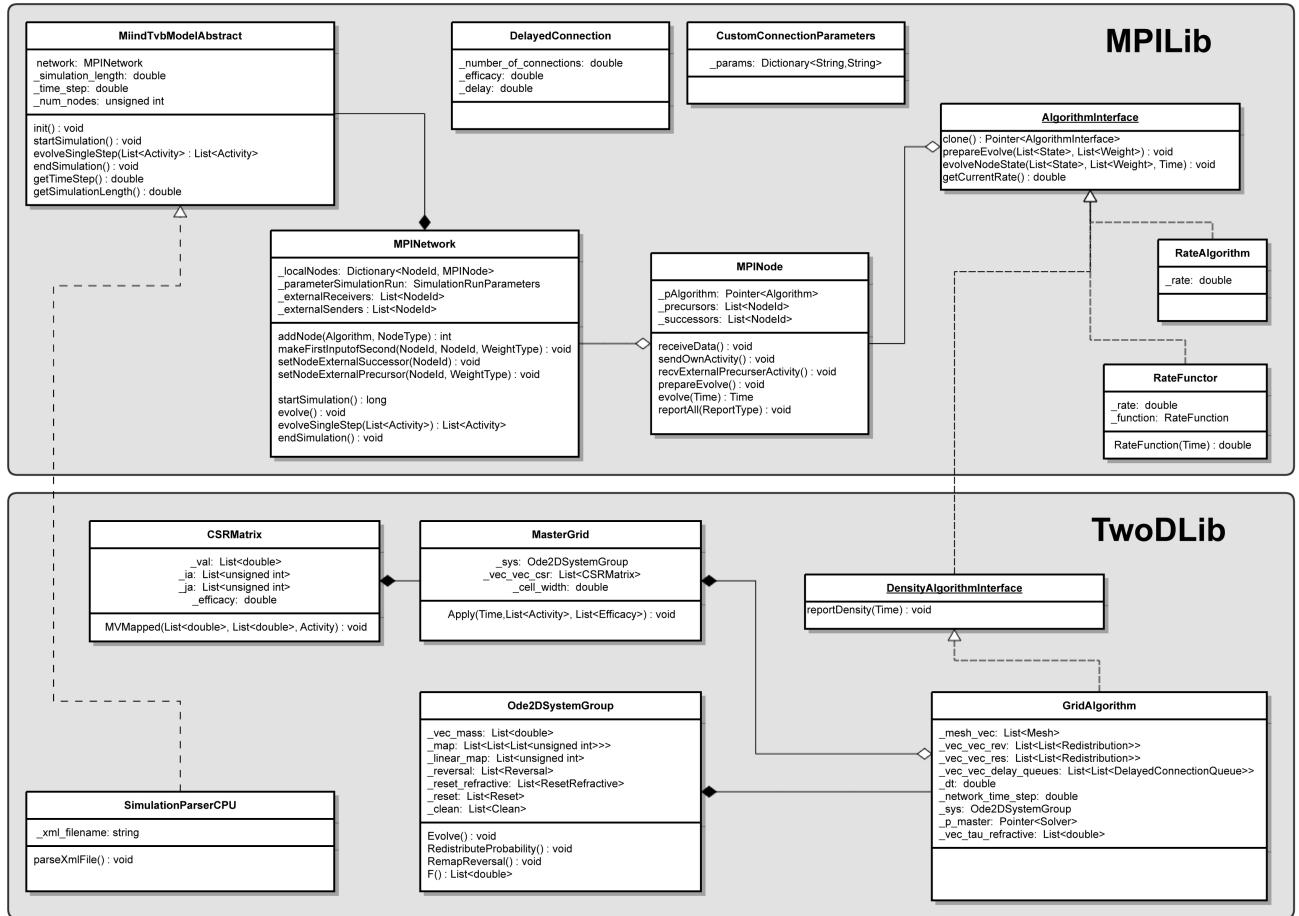
**Figure 9.** Heat plots for the probability density functions of two populations in MIIND. Brightness (more yellow) indicates a higher probability mass. Scales have been omitted as the underlying neuron models are arbitrary. (A) When the Poisson master equation is solved, probability mass is pushed 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.



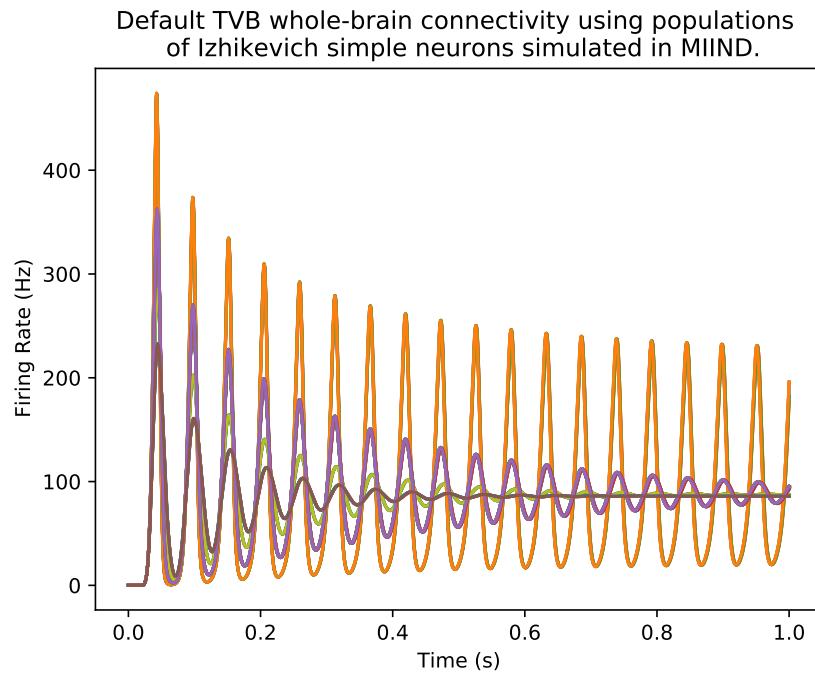
**Figure 10.** The MIIND processes and generated files required at each stage of pre-processing for the mesh algorithm. The shaded green rectangles represent automated processes run via the MIIND CLI.



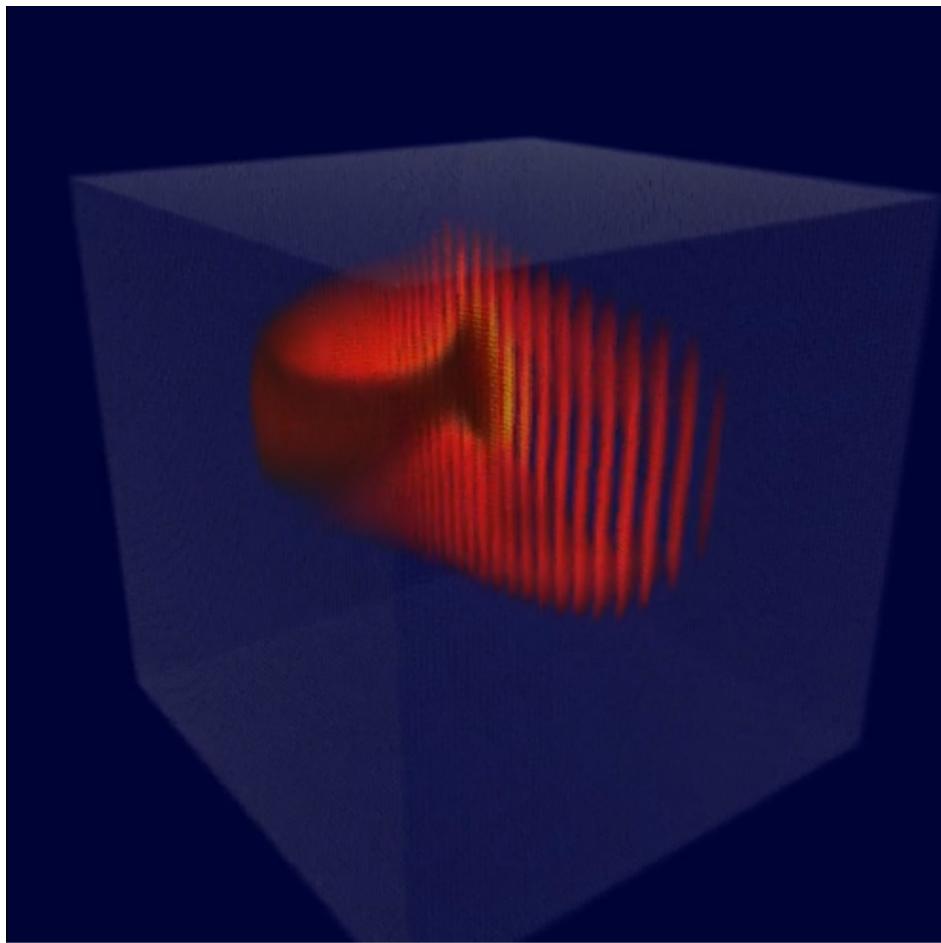
**Figure 11.** (A) The average firing rate of a population produced by calling the **rate** command. (B) A density plot (normalised to the maximum density value) of the population produced by calling the **plot-density** command. (C) The marginal density plots produced by calling the **plot-marginals** command.



**Figure 12.** A minimal UML diagram of MIIND. The two major libraries, MPILib and TwoDLib, are represented.



**Figure 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 remain in an oscillating state.



**Figure 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.