

# CNS tutorial MIIND

09.04.2019

---

Hugh Osborne, Lukas Deutz, Marc de Kamps

School of Computing

University of Leeds

Leeds LS9 2JT

United Kingdom

## Overall Objective

On the 13 July 2019, we will run a full day tutorial at the Computational Neuroscience 2019 conference (CNS2019), with the aim of providing an introduction to modelling brain dynamics using population density techniques. The overall aim is to introduce an overview of these techniques, and to provide hands on experience in using MIIND, a simulator that allows the user to make models of neural circuits at the population level. We will demonstrate practical use cases: simulations of single populations of leaky- and quadratic-integrate-and-fire (LIF, QIF) neurons, as well as two dimensional model neurons, e.g. adaptive-exponential-integrate-and-fire (AdExp). We will also consider circuit models of such population. This part of the session will replicate some well known papers on population level modeling. We will introduce an XML-based format in which users can define their own circuits and discuss the workflow to create, run and analyse these simulations. Finally, we will show how to incorporate novel models into the simulator, by demonstrating how one create one's own 2D model. This enables the user to study 2D dynamical models (not just neuronal ones), subject to noise.

## Learning Outcomes

At the the end of this workshop, the user should

1. Have a comprehensive overview of what population density techniques can and cannot model.
2. Have an overview of differences and commonalities with other population level modelling techniques, such as rate-based models and others.
3. Have an overview of installation methods for MIIND: Docker, package, source code.
4. Be able to run stock provided simulation scripts in XML through the MIIND simulator.
5. Be able to modify stock simulation XML scripts, and tailor it to the user's purposes.
6. Be aware of the tools for analysing simulation results, both within MIIND and through third party tools such as Numpy/Scipy.
7. Be aware of the different ways of parallelizing MIIND generated code, in particular on GPGPU.
8. Know how to add novel models, and have a good idea how to study two dimensional dynamical systems subject to noise in general.

## Syllabus

*Introduction to the theory of population density methods (PDMs):* population level modelling; where do PDMs sit compared to spiking neuron simulations? What are the underlying assumptions - when are they reasonable, when not? What are the advantages/disadvantages compared to other methods?

*Example: leaky-integrate-and-fire neurons (LIF):* interaction between neural and stochastic dynamics; What is the Master equation of Poisson process? How do we model Gaussian white noise? What is the relation between steady states and transfer (gain) curves? What is the linear response approximation? How can we model this in MIIND?

*Example: quadratic-integrate-and-fire (QIF):* What is a geometric grid? What are the commonalities and differences between LIF and QIF? How is the QIF used to model a burster?

*Setting up simulations using stock provided models:* XML simulation file; editing neural and network parameters; generating C++/Cuda code; running and monitoring a simulation; where are the simulation results? How can I analyse them?

*Some two-dimensional models:* What is a 2D model? How do I represent it by a Mesh (or Grid)? What are matrix files and why do I need them? How do I adapt a simulation XML file to incorporate a 2D model?

*Example: adaptive-exponential-integrate-and-fire (AdExp):* What are striking differences between LIF neurons with and without adaptation? What does the two dimensional density look like? How can I get marginal distributions?

*Parallelisation:* Why parallelise? What should I use, OpenMP or Cuda? Can I use Cuda on the HBP machines?

*Circuits:* How do I build larger networks? How can I incorporate delays between populations?

*Example:* balance of excitation-inhibition with and without adaptation, ...

## Resources

1. We strongly recommend that users bring a laptop with a pre-installed version, but we will reserve half an hour to help with installation problems.
2. PDF versions of the following papers:

- 
- [A. Omurtag, B.W. Knight, L. Sirovich \(2000\). On the Simulation of Large Populations of Neurons](#)
  - [N. Fourcaud-Trocmé, D. Hansel, C. van Vreeswijk and N. Brunel \(2003\). How Spike Generation Mechanisms Determine the Neuronal Response to Fluctuating Inputs](#)
  - [D. J. Amit and N. Brunel \(1997\). Model of global spontaneous activity and local structured activity during delay periods in the cerebral cortex](#)
  - [M. de Kamps, M. Lepperød, Y. M. Lai \(2019\). Computational geometry for modeling neural populations: From visualization to simulation](#)

# MIIND Basic Workflow

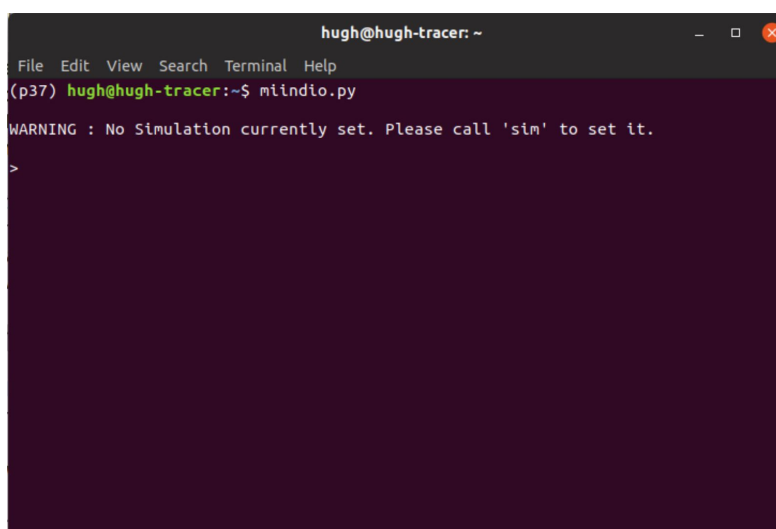
## Building and Running a Simulation

To explore the core MIIND workflow, we will set up a simple simulation of a single population of neurons then briefly analyse the results. Just for fun, we won't reveal the specific underlying neuron model we're using until the end. Hopefully, you will recognise what it is as we start working with it but to understand the workflow, it is not necessary to know. All the required files for this tutorial have been included in the MIIND installation. If MIIND was installed to system, the files can be found in the *<Install Location>/share/miind/examples/tutorial*. If MIIND was built "in place", they are in *<MIIND Source Directory>/examples/tutorial*. In the docker, these files are available in */usr/share/miind/examples/tutorial*.

Copy the following files from *tutorial/t1* into a working directory of your choice:

<b>t1.mesh</b>	<b>t1_0.1_0_0_0.mat</b>	
<b>t1.model</b>	<b>t1.xml</b>	<b>t1.projection</b>

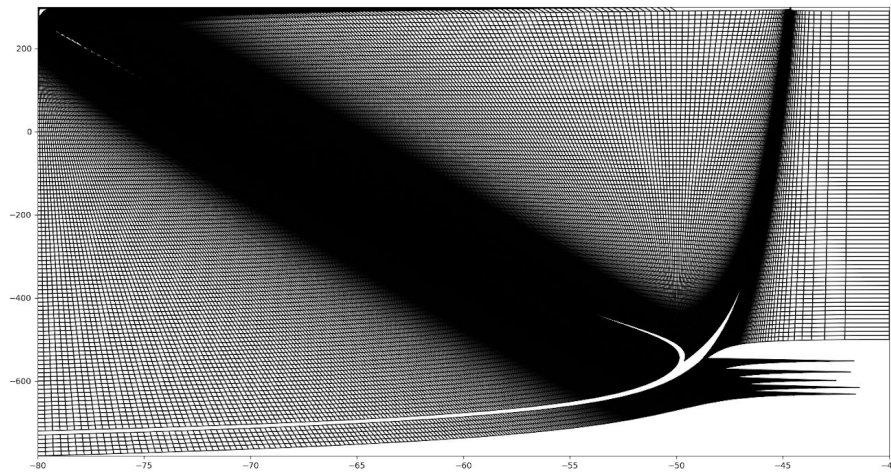
The *t1.mesh* file contains the vertices of the mesh which describes the deterministic dynamics of the neuron model. Although the *t1.mesh* file itself is not important for the simulation, it was used to generate the *t1.model* file which contains a copy of the mesh. In your working directory, open a terminal window and call `miindio.py`.

A terminal window titled 'hugh@hugh-tracer: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is '(p37) hugh@hugh-tracer:~\$' and the command 'miindio.py' has been entered. The output shows a warning message: 'WARNING : No Simulation currently set. Please call 'sim' to set it.' followed by a prompt '>'.

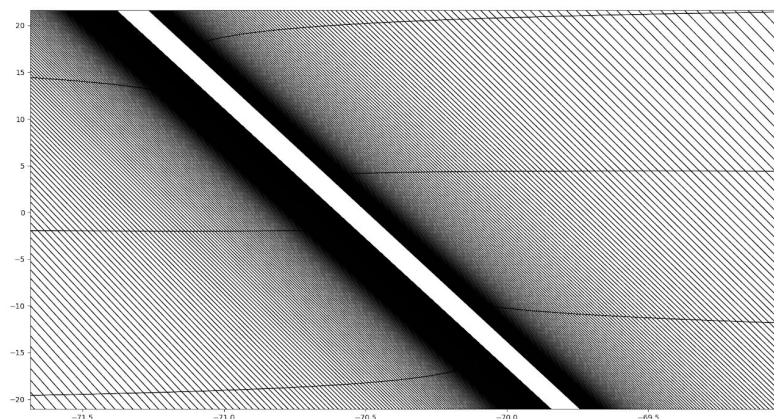
```
hugh@hugh-tracer: ~
File Edit View Search Terminal Help
(p37) hugh@hugh-tracer:~$ miindio.py
WARNING : No Simulation currently set. Please call 'sim' to set it.
>
```

miindio.py produces a command line interface for working with MIIND simulations. Let's use the following command to take a look at the mesh.

**> draw-mesh t1.mesh**



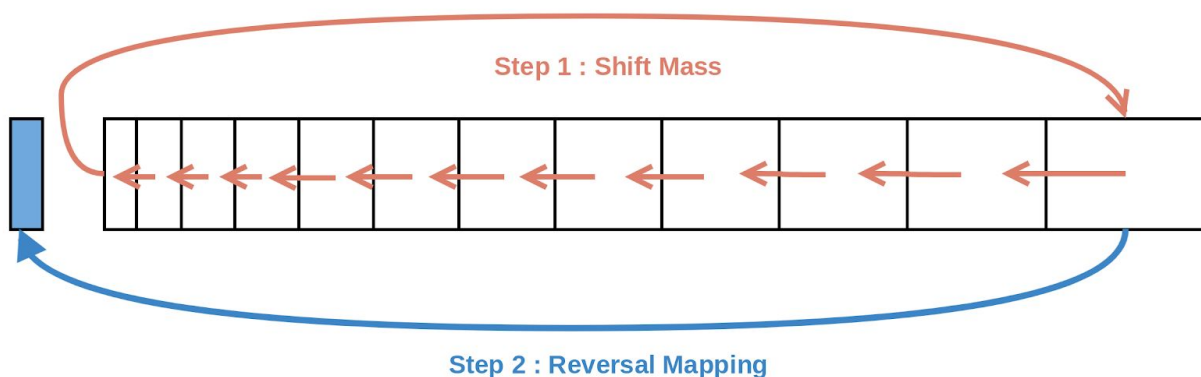
The mesh is made up of multiple strips of cells which follow the characteristic curves of the neuron model. During the simulation, each cell will contain probability mass which will be transferred to the next cell along each strip after a single timestep. Most neuron models, including this one, have membrane potential as a time dependent variable and this is represented on the x axis. We will see later the meaning of the variable along the y axis. There is a stationary point at  $(-70.7, 4.0)$ . Use the zoom and pan tools in the plot window to find the approximate location of the stationary point.





Most of the strips approach this point and as they do so, the dynamics slow down and so the cells become very small. There is a stationary cell which is not shown in the plot but which covers the stationary point. Probability mass, once it reaches the end of the approaching strips, is transferred to this stationary cell to avoid the need for infinitesimally small cells.

Open the `t1.model` file. This is the actual file which MIIND uses to read in the mesh data as well as some additional mapping information. You can see that it has an XML format and the first child of the Mesh object is `<TimeStep>`. This value is the amount of time in seconds which mass spends in each cell before shifting to the next down the strip. Search the text for `'Mapping type="Reversal"'`. Each line in the reversal mapping has three values: the source cell coordinates, the destination cell coordinates, and the proportion of mass to be transferred. The reversal mapping is most often used to define how mass is transferred from a strip to a stationary cell. At the stationary point of our neuron model, we wish to transfer mass from the end of each strip to the stationary cell. MIIND's algorithm works by first performing the shift of mass one cell down the strip then performing the reversal mapping. Mass which is at the end of a strip is therefore shifted back to the beginning of the strip before it can be transferred to the stationary cell. In order to correctly transfer the mass, we introduce the reversal mapping to, in fact, take mass from the first cell in the strip to the stationary cell. The coordinates are made up of the strip number followed by the cell number. By default, the coordinates of the stationary cell is 0,0. So for a given strip, we can see a transition from the first cell in each strip ( $n, 0$ ) to  $(0, 0)$  and the entirety of the mass is transferred (1.0).



Below the reversal mapping section are two single values: `threshold` and `V_reset`. This is a hint that our mystery model has a threshold/reset functionality. The `'Mapping type="Reset"'` section below that provides the actual implementation of that functionality. As with the reversal mapping, each line is a transition of a proportion of mass from a source cell to a target cell. In the MIIND algorithm, the reset transitions are performed at the end of each time step. The `t1.model` file can now be closed.

We will not spend time here looking in depth at the *t1\_0.1\_0\_0\_0.mat* file but it contains further transitions of proportions of mass between cells in the mesh when the neurons receive a single spike. The second value in the filename (0.1) represents the instantaneous efficacy on the membrane potential due to a single incoming spike. MIIND provides an automated way to generate these files and one file must be generated for each expected efficacy to be used in the simulation.

Back in the working directory, open the *t1.xml* file for editing. This file will contain the description of our simulation but it is currently unfinished. There are five main sections to the MIIND simulation file along with various smaller properties, all defined within the `<Simulation>` tags.

The `<Algorithms>` section contains definitions for how our population or populations will behave. First add the following code to the Algorithms section:

```
<Algorithm type="MeshAlgorithm" name="T1_ALG" modelfile="t1.model" >  
<TimeStep>0.0002</TimeStep>  
<MatrixFile>t1_1_0_0_0.mat</MatrixFile>  
</Algorithm>
```

This definition says that we will be simulating populations of neurons using MIIND's Population Density Technique which has a type="MeshAlgorithm". We give the algorithm a name ("T1\_ALG") as a unique identifier. We also reference the *t1.model* file. If we were to require populations with different underlying neuron models, we would define additional MeshAlgorithms with different model files. The TimeStep should match the value used to generate the mesh and which was quoted in the model file. Finally, any number of `<MatrixFile>`s can be added to this definition for each required efficacy to be used with this model. In this case, we will only have a single input to our population with an efficacy of 0.1. We also need another Algorithm definition, to provide Poisson distributed input with an average rate.

```
<Algorithm type="RateFunctor" name="Exclnput">  
<expression>2500.</expression>  
</Algorithm>
```

This will produce a constant average firing rate of 200 Hz. The expression here also allows for expressions in terms of *t* which represents the simulation time.

The next section is `<Nodes>`. In here, we need to define the actual instances of the Algorithms. In many cases we have multiple populations using the same underlying neuron model and algorithm (for example MeshAlgorithm) so multiple nodes can reference a single algorithm. To the `<Nodes>` section, add the following lines.



```
<Node algorithm="T1_ALG" name="Pop" type="NEUTRAL" />
<Node algorithm="ExcInput" name="Drive" type="EXCITATORY_DIRECT" />
```

These two lines indicate that we want a single population named “Pop” using the T1\_ALG algorithm and a single input named “Drive”. The type attribute describes the form of the output of the population (node). Pop is set to NEUTRAL indicating that it can produce both excitation or inhibition to other target populations (of course, there are no target populations in this simulation). The Drive node is set to EXCITATORY\_DIRECT which means that target populations should expect a Poisson distributed input with post-synaptic excitatory potentials (positive efficacies) only.

The <Connections> section is where we will define how our nodes are connected. Add the following lines to the <Connections> section.

```
<Connection In="Drive" Out="Pop">1 1 0</Connection>
```

There is only a single connection between the input (In) set to Drive and the target neuron population (Out) set to Pop. The three parameters are:

- The number of input connections to each theoretical neuron in the target population.
- The post-synaptic potential (efficacy)
- The transmission delay

In this case, every theoretical neuron in the Pop population has ten incoming connections. So each neuron theoretically receives Poisson distributed spikes with a rate of 200Hz through each of the ten connections for a total input rate of 2Hz \* 10 connections. This model measures membrane potential in mV so a single spike causes the membrane potential to jump by 0.1mV. Finally, there is no delay between Drive and Pop.

At this point, the network for the simulation has been completely defined. What is left is to set the sort of reporting which MIIND will produce and some parameters for the simulation itself. In the <Reporting> section, there are three possible ways to observe the MeshAlgorithm node. Add the following lines to the <Reporting> section.

```
<Density node="Pop" t_start="0.0" t_end="1.0" t_interval="0.0002" />
<Rate node="Pop" t_interval="0.0002"/>
<Display node="Pop" />
```

The first option is to write the entire density profile out from the population using the <Density> tag. You can set the target node which must be using a MeshAlgorithm. t\_start and t\_end define when to start and stop recording and t\_interval indicates how often to

write a file. Each density file is quite large so having a small interval can lead to slower simulations and a lot of used disk space.

The <Rate> tag produces a single file with each line recording the time and the average firing rate of the named node every t\_interval. There are no t\_start or t\_end attributes as these files are generally fairly lightweight unless the t\_interval is set very small.

Finally, the <Display> tag will render the mesh and current density to the screen every timestep during the simulation. It can be useful to observe the population as it evolves during the simulation but it does slow the completion time considerably. The rendered plot is also saved as an image file every timestep and is therefore, like <Density>, rough on the hard drive but these images can be used to create movies. For this small demonstration, we can enable all three forms of recording without a huge toll.

The last section is <SimulationRunParameters> which holds some simple parameters to the simulation. Add the following lines to this section.

```
<SimulationName>t1</SimulationName>  
<t_end>0.35</t_end>  
<t_step>0.0002</t_step>  
<name_log>t1.log</name_log>
```

Here we can set the name of the simulation, the time duration in seconds (in this case, just 1 second), the time step of the simulation which must be a multiple of the time step defined by the MeshAlgorithm, and the filename for the log file to catch any errors. Additional options for all of the above sections and in depth descriptions of all the MIIND XML features will be provided in the upcoming documentation.

Everything is now ready to build and run the simulation. Close the t1.mesh plot if you haven't already and go back to the miindio command line interface. Miindio keeps track of a "current working simulation" within each directory it is run. When you first ran miindio.py, you may have noticed a file miind\_cwd was created in the directory. We can set the current working simulation by using the sim command. If there were any problems with the xml file, MIIND will give a notification.

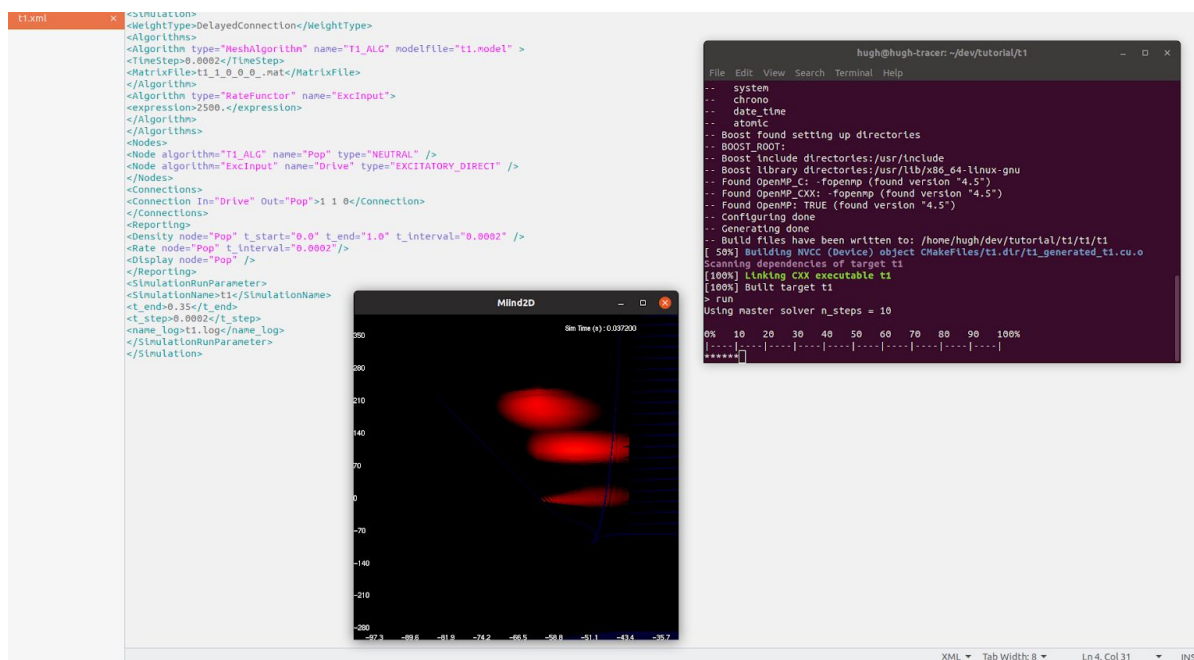
**> sim t1.xml**

Now, if we close the command line interface and reopen it in the same directory, MIIND will remember that we are working with the t1.xml simulation. If you just type sim without an xml file, you will be presented with general information about the simulation. The next step is for MIIND to translate the xml file into a C++ source file and compile it into an executable. To do this, use the submit command.

**> submit**

If all is well, you should see the output from the cmake command which is internally called from MIIND. You may also notice that an output directory has been added to the working directory which contains the generated code and built executable. The model and mat files have also been copied into this directory. To run the simulation, call run.

> run



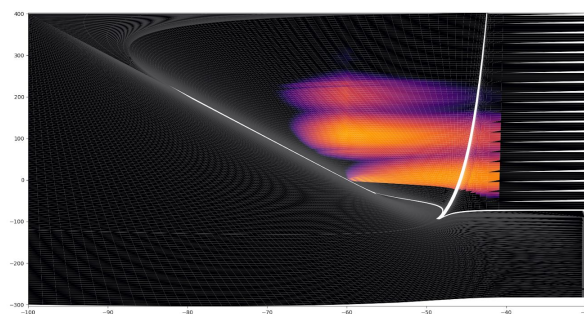
Because we included the `<Display>` tag in the `<Reporting>` section of the xml, you should be able to watch the probability density function evolve over time. In the output directory, there is also a new directory holding the recorded density files, a directory holding the display images, and a file `rate_0` holding the firing rates of the Population. Having watched the density change over time, you may now recognise which neuron model we have used. Not to worry if you haven't. In the next section we will use the command line plotting tools to learn more about it.

## Analysing the Simulation Results

The output metrics from the simulation are the density profiles for each `t_interval` which list the amount of probability mass in each cell of the mesh, and the rate files which hold the average firing rate of the population at each `t_interval`. These files are in ASCII format and can be read into your favourite analysis pipeline. The MIIND python API upon which `miindio` is built can also be used to bring these results into python. The API will be introduced in a later tutorial. For now, we shall simply use the command line interface to

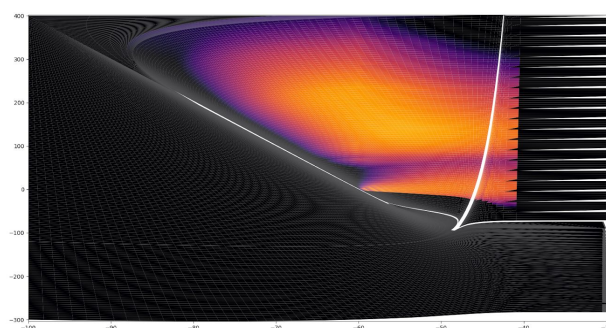
plot the results and hopefully learn more about the mystery neuron model. In the command line interface, call the `plot-density` command to view the density profile at a given point in the simulation.

**> plot-density Pop 0.01**



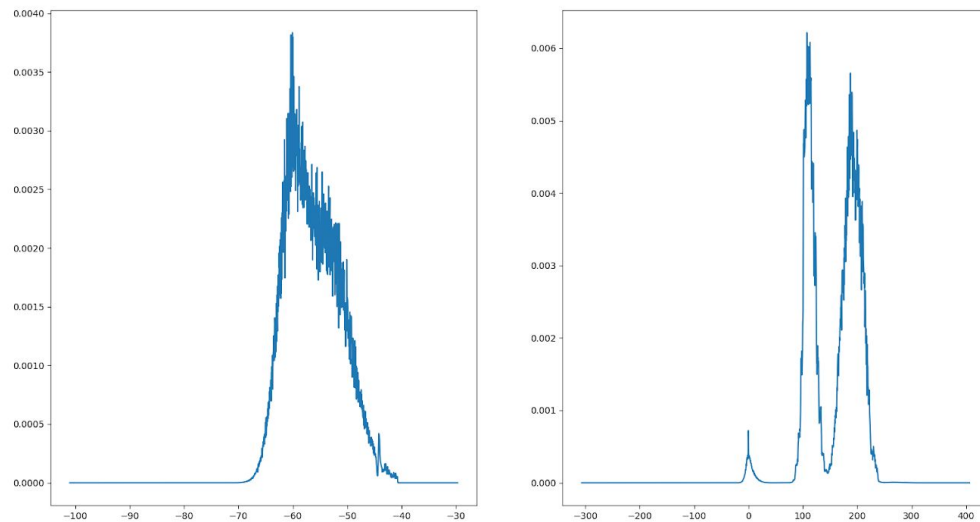
This density plot shows a hidden feature which was not apparent when we looked at the mesh and model files but could be seen in the display during simulation. The total mass initially started at the stationary cell and has been pushed towards the reset threshold due to the input from Drive. However, instead of simply being moved to the reset potential with the same value in the y direction, the mass has been shifted upwards as well. Let's look at the stationary state of the density later in the simulation.

**> plot-density Pop 0.34**

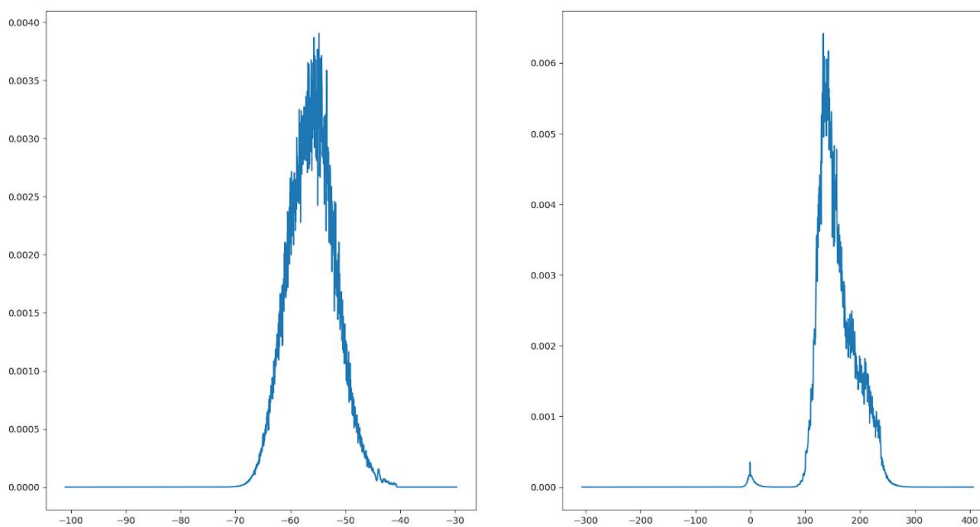


The distinct layers have now merged into a single area of state space. The shape of the population is such that the higher the y value, the further back from the threshold the mass sits. This indicates that, at higher values of y, it is more difficult for neurons to reach the threshold. We can also look at the one dimensional marginal densities. Those are the densities as observed in the two dimensions separately. Use the `plot-marginals` command to do this.

```
> plot-marginals Pop 0.05
```

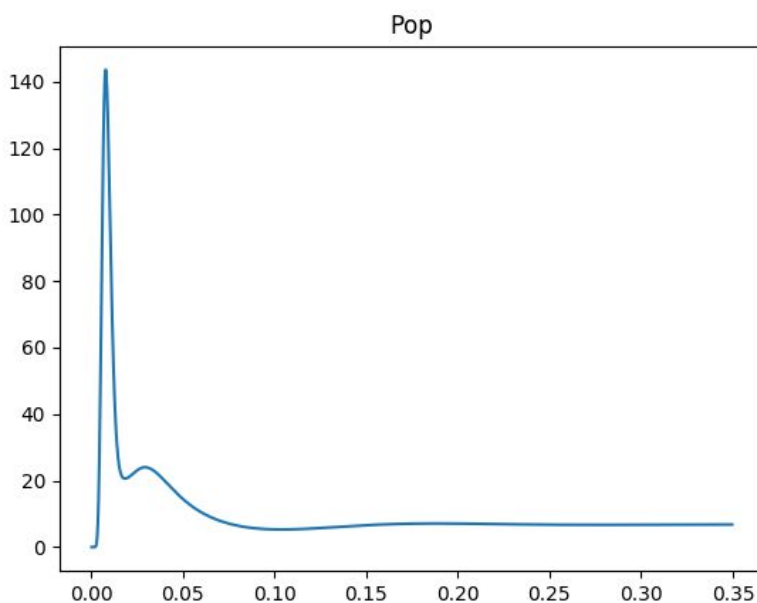


```
> plot-marginals Pop 0.34
```



Early in the simulation, the distinct layers of mass are also clearly visible in the y axis marginal. Finally, let's look at the output firing rate of the population by using the rate command.

> rate Pop



There is an initial transient and a high firing rate to start with. If this were exponential integrate and fire, it would remain at that higher rate but in this case, the population is made up of adaptive exponential integrate and fire neurons which after a short period of time, adapt to the input and relax to a lower firing rate. The y axis, the second time dependent variable in the model is an adaptive variable which makes the neuron less likely to fire for higher values and more likely to fire for lower values. If you go back to view the mesh (draw-mesh t1.mesh), you can see this feature in the wider cells between the reset and threshold potentials at higher values of the adaptive variable. Wider cells (in the x direction) mean that neurons lose membrane potential quicker.

As we used the <Display> tag in <Reporting>, we can use the generate-density-movie command to create a movie and re-watch the simulation. Alternatively, we can just re-call run in miindio.

> generate-density-movie Pop 512 0.1 adex

This command will generate an mp4 movie of the Pop density called adex.mp4 in the same directory as the xml file with a frame size of 512 pixels at 10 times the simulation time which in this case will be 10 seconds.



## Challenge

There are two more mystery neuron models available in **.../tutorial/t2** and **.../tutorial/t3**. Simulate populations of these neurons in MIIND to discover what they are. Whether you recognise the models or not, produce some plots and movies to demonstrate the behaviour of these populations.

# Vectorised GPGPU and GridAlgorithm

In this tutorial, we will look at how MIIND can use a cuda enabled graphics card to perform a massively parallelised implementation of your simulation. We will also discover a slightly different method of simulating each population without the need for building a complex mesh and use this approach to build a so called half-centre oscillator network. If your system does not support cuda, move straight to the Grid Method section.

## Vectorising on the GPU

In order to parallelise the MIIND population density technique algorithm so it can be run on the GPU, all meshes (one from each population in the network) are flattened to a single one dimensional vector and stitched together. The vector is passed to the graphics card and processing then occurs on large numbers of cells of that vector in parallel: moving mass from one cell to the next; spreading the mass across cells due to the input by solving the master equation; performing the reversal and reset mapping. We will not go into detail about the process under the hood but demonstrate how easy it is to change your XML simulation to switch to a vectorised format. If you have a machine which supports CUDA and MIIND was installed with CUDA enabled, you should be good to go.

Copy *tutorial/vectorised/* to your workspace and open *vector.xml*. You'll see that there are two ADEX populations, one excitatory and one inhibitory. There are connected together and there is an external drive to the excitatory population. Currently, the simulation is set up to run in the normal way on the CPU. However by changing the MeshAlgorithm to MeshAlgorithmGroup, MIIND will now build the simulation executable as a CUDA enabled vectorised format.

```
<Algorithm type="MeshAlgorithmGroup" name="ADEX_ALG" modelfile="adex.model" >
<TimeStep>0.0002</TimeStep>
<MatrixFile>adex_1_0_0_0.mat</MatrixFile>
<MatrixFile>adex_-1_0_0_0.mat</MatrixFile>
</Algorithm>
```

This is all that is required to switch seamlessly between CPU and GPU based approaches. Depending on your architecture, you should see a speed increase to the simulation using the GPU version.

## GridAlgorithm

It can sometimes be difficult to produce the desired mesh for a population's underlying neuron model. If there are stiff dynamics, large differences in speed, or areas of state space which are difficult to cover, producing the mesh can be a time consuming process (although a big upside is that it ensures full understanding of the dynamics of the model). One solution is to use the GridAlgorithm in MIIND. Instead of using a mesh which follows the deterministic dynamics of the model, a regular grid is defined across the entire state space. An euler process is used to define how mass moves from one cell to other cells due to the dynamics in one timestep. This is a similar technique to the generation of the matrix transition file. During simulation, instead of moving mass down strips, the transition matrix is applied once per timestep. Copy tutorial/grid/ to your workspace. As we work through an example, you will see a number of advantages to using GridAlgorithm. Open izhikevich.py for editing. This python script defines the izhikevich model in a method which returns the time derivatives of the model based on the given values, similar to other numerical solvers. In this case, we have defined the izhikevich model which we saw earlier in mesh form.

```
def izh(y,t):
    v = y[0];
    w = y[1];

    v_prime = 0.04*v**2 + 5*v + 140 - w + 10
    w_prime = 0.02 * (0.2*v - w)

    return [v_prime, w_prime]
```

When the script is run, a call is made to grid\_generate which will produce the required files. The parameters in order are: the name of the function, the timestep to use, the timescale, a tolerance value for the solver, the desired basename of the generated files, the threshold potential, the reset potential, an optional vertical reset shift (like in adex), the minimum and maximum membrane potential, the minimum and maximum second variable. The two final values define the resolution of the grid.

```
grid_generate.generate(izh, 0.1, 0.001, 1e-4, 'izh', -30.0, -50.0, 2.0, -85.0, -10.0, -15.0, 20.0,  
500, 500)
```

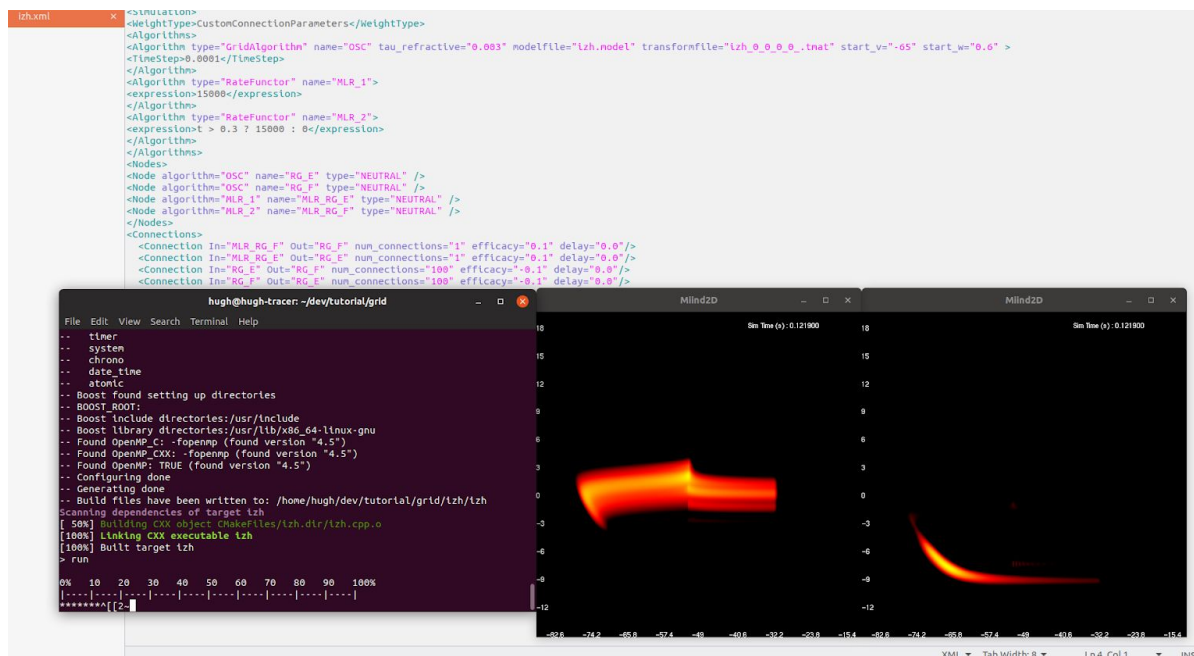
Close the python file and run the script in a terminal window.

## python izhikevich.py

The two files generated are *izh.model* and *izh.tmat*. The model file is very uninteresting as it just holds a regular grid and the reset mapping. The tmat file is what describes the movement of mass due to the deterministic dynamics. We have seen that one advantage to using GridAlgorithm is that there is no need for a user built mesh. Another advantage is that, because of the regularity of the grid, spike efficacies can be calculated on the fly during simulation avoiding the need to predefine them. Open the izh.xml file for editing. Add the following lines to the Algorithms section.

```
<Algorithm type="GridAlgorithm" name="OSC" tau_refractive="0.003"
modelfile="izh.model" transformfile="izh_0_0_0_0.tmat" start_v="-65" start_w="0.6" >
<TimeStep>0.0001</TimeStep>
</Algorithm>
```

This is how the GridAlgorithm is defined. Like MeshAlgorithm, it requires a name, modelfile and optional tau\_refractive value as well as a TimeStep child value. However, you must also provide a reference to the tmat file and give the coordinates of the starting position in state space. The rest of the simulation has been set up for you. It consists of two populations of izhikevich neurons which are mutually inhibiting each other. Note that any value is allowed as an efficacy in the connection definitions without the need for matching matrix files. Run miindio.py, set the simulation to izh.xml, call submit and run.



Note that you can also utilise the vectorised approach for GridAlgorithm by changing the Algorithm type to GridAlgorithmGroup.

## Challenge

Try recreating the izhikevich grid with a different grid resolution or with different parameters in the izh function and see how it affects the simulation.

Come up with your own neuron model and create a new xml file to simulate a population.