Creating New Neuron Models for SpiNNaker

Introduction

This manual will guide you in the creation of new neuron models to be run on SpiNNaker. This includes the C code that will be compiled to run on the SpiNNaker hardware, as well as the Python code which interacts with the PyNN script to configure the model.

Installation

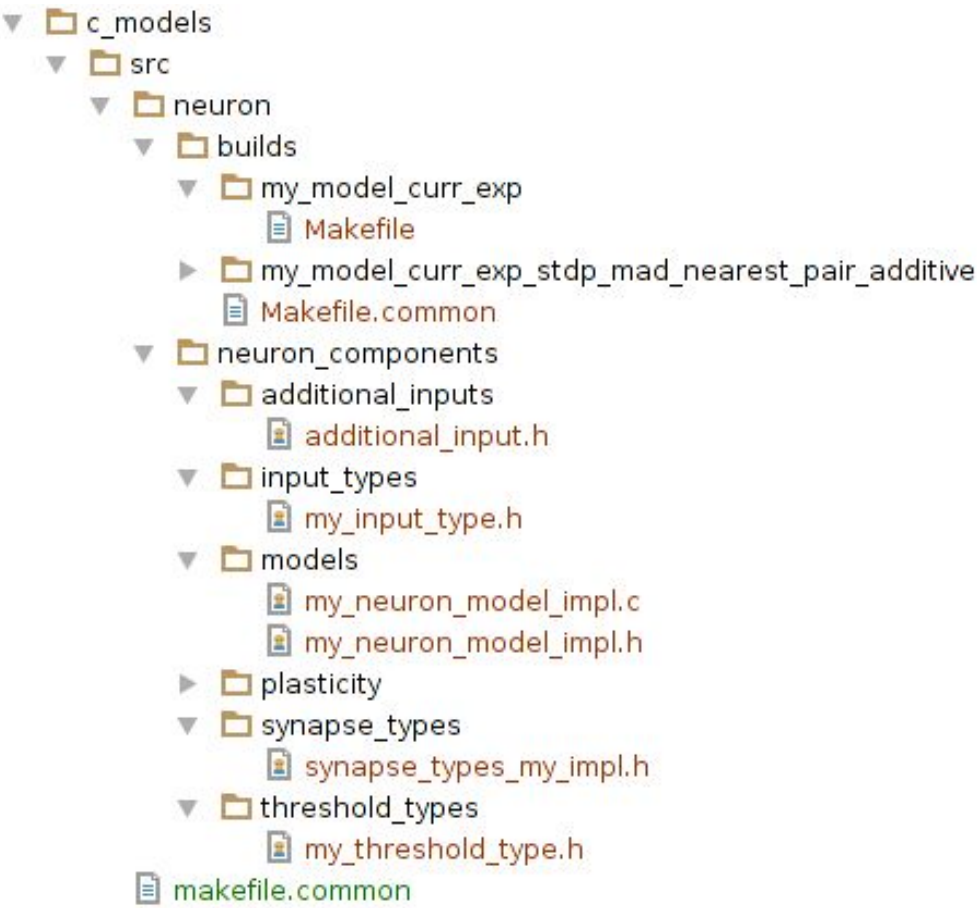
In order to create new models, you will need to ensure that you have set up a development environment suitable for compiling C code for SpiNNaker. This can be done by following the instructions here:

http://spinnakermanchester.github.io/2016.001.AnotherFineProductFromTheNonsenseFactory/spynnaker_pages/PyNNOnSpiNNakerExtensions.html

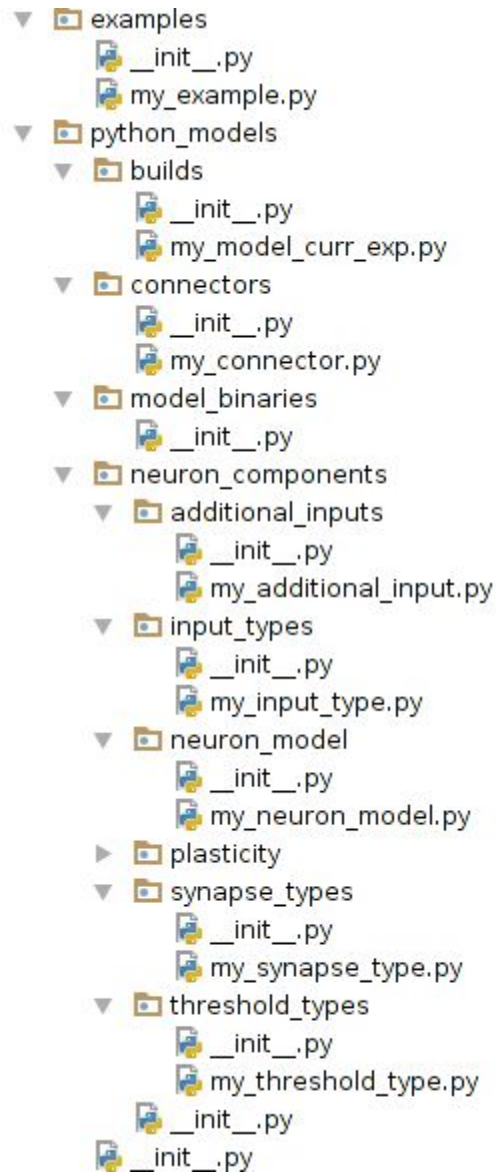
Project Layout

The recommended layout for a new model project is shown below; this example shows a model called "my_model", with current-based exponential synapses. It is recommended that the C and Python code is kept in the same project to help keep them synchronized.

C code



Python code



This template structure can be downloaded from:
http://spinnakermanchester.github.io/2016.001.AnotherFineProductFromTheNonsenseFactory/workshop_material/supplementary_material/template_new_model.zip

C model builds

All neuron builds consist of a collection of components which when connected together produce a complete Neural Model. These components are defined in **Table 1**:

Component	**Definition**
Input component.	The type of input the model takes. Currently there is conductance and Current based inputs supported by sPyNNaker.
Synapse Type component.	The synapse type controls the shaping of the synapses in response to the input weights. Within sPyNNaker, support so far includes exponential synapses (with one excitatory and one inhibitory synapse per neuron which decay exponentially with a configured time-constant) and dual-excitatory exponential synapses (with 2 separate excitatory synapses and one inhibitory synapse per neuron, decaying as per the previous type).
Threshold component.	The way a neuron decides to fire. Currently this is a static value. But during the labs, you will be building a stochastic version.
Additional input component.	Any other parts of a input (currently this has no implementations)
Neuron model component.	The underlying behaviour of the neuron model. Currently there is IZK, leaky integrate and leaky integrate and fire versions supported by sPyNNaker.

Table 1: Different supported components

Each build is stored within its own folder in the c_models->src->neuron->builds directory. Within each build is a c *Makefile* which describes the separate components required to build that specific form of Neuron model.

If we look at the simple my_model_curr_exp's makefile located in: c_models->src->neuron->builds->my_model_curr_exp->Makefile (as shown in **code 1)**, we can see that:

```
1 .  APP = $(notdir $(CURDIR))
2 .  BUILD_DIR = build/
4 .  NEURON_MODEL = $(SOURCE_DIRS)/neuron/neuron_components/models/neuron_model_lif_impl.o
5 .  NEURON_MODEL_H = $(SOURCE_DIRS)/neuron/neuron_components/models/neuron_model_lif_impl.h
6 .  INPUT_TYPE_H = $(SOURCE_DIRS)/neuron/neuron_components/input_types/input_type_current.h
7 .  THRESHOLD_TYPE_H = $(SOURCE_DIRS)/neuron/neuron_components/threshold_types/threshold_type_static.h
8 .  SYNAPSE_TYPE_H = $(SOURCE_DIRS)/neuron/neuron_components/synapse_types/synapse_types_exponential_impl.h
9 .  SYNAPSE_DYNAMICS = $(SOURCE_DIRS)/neuron/neuron_components/plasticity/stdp/synapse_dynamics_static_impl.o
11 . include ../Makefile.common
```

Code 1: my_model_curr_exp's Makefile

1. On line 1 we are declaring that any files made will be based from the current directory.
2. On line 2 we are declaring that this model is to be compiled in a folder called build.
3. Lines 4 and 5 state that for neuron model component (described in **Table 1)** used for this model build, are located in the models folder within the neuron_components under the name *neuron_model_lif_impl*.o /.h (both are needed)
4. Line 6 states that for this model build, the Input type component (described in **Table 1)** is located within the input_types folder within the neuron_components under the name input_type_current.h (no object file is needed here).
5. Line 7 states that for this model build, the Threshold type component (described in **Table 1)** is located within the threshold_types folder within the neuron_components under the name threshold_type_static.h (no object file is needed here).
6. Line 8 states that for this model build, the synapse type component (described in **Table 1)** is located within the synapse_types folder within the neuron_components under the name synapse_types_exponential_impl.h (no object file needed here).
7. Line 9 states that for this model build, the synapse dynamics component (described in **Table 1)** is located within the plasticity->stdp folder within the neuron_components under the name synapse_dynamics_static_impl.o (no .h file needed here).
8. Line 11 tells the make system to import the next level up Makefile so that it can detect where the rest of the code needed to be linked in can be found.

If we now look at the my_model_curr_exp_stdp_mad-nearest_pair_addive's makefile, as shown in **code 2.** We can see that this build is the same as my_model_curr_exp but also includes components required for plasticity. This is shown through:

```
1 .     APP = $(notdir $(CURDIR))

2 .     BUILD_DIR = build/

4 .     NEURON_MODEL = $(SOURCE_DIRS)/neuron/neuron_components/models/neuron_model_lif_impl.o

5 .     NEURON_MODEL_H = $(SOURCE_DIRS)/neuron/neuron_components/models/neuron_model_lif_impl.h

6 .     INPUT_TYPE_H = $(SOURCE_DIRS)/neuron/neuron_components/input_types/input_type_current.h

7 .     THRESHOLD_TYPE_H = $(SOURCE_DIRS)/neuron/neuron_components/threshold_types/threshold_type_static.h

8 .     SYNAPSE_TYPE_H = $(SOURCE_DIRS)/neuron/neuron_components/synapse_types/synapse_types_exponential_impl.h

9 .     SYNAPSE_DYNAMICS = $(SOURCE_DIRS)/neuron/neuron_components/plasticity/stdp/synapse_dynamics_stdp_mad_impl.o

10 . TIMING_DEPENDENCE = $(SOURCE_DIRS)/neuron/neuron_components/plasticity/stdp/timing_dependence/timing_nearest_pair_impl.o

11 . TIMING_DEPENDENCE_H = $(SOURCE_DIRS)/neuron/neuron_components/plasticity/stdp/timing_dependence/timing_nearest_pair_impl.h

12. WEIGHT_DEPENDENCE =
$(SOURCE_DIRS)/neuron/neuron_components/plasticity/stdp/weight_dependence/weight_additive_one_term_impl.o

13. WEIGHT_DEPENDENCE_H =
$(SOURCE_DIRS)/neuron/neuron_components/plasticity/stdp/weight_dependence/weight_additive_one_term_impl.h

15 . include ../Makefile.common
```

Code 2: my model curr exp stdp mad nearest pair additive Makefile

1. Lines 1 to 8 are the same as in **Code 1**.
2. Line 9 informs the makefile that for this build, the synapse dynamics component (described in **Table 1)** is located within the plasticity->stdp folder within the neuron_components under the name synapse_dynamics_stdp_mad_impl.o .This difference from synapse_dynamics_static_impl.o implies that this build requires plasticity and therefore the model requires a Timing and Weight dependence component.
3. Lines 10 and 11 define the location of the .h and .o file for Timing dependence component required by this model build. In this case it is located in plasticity/stdp/timing_dependence folder from neuron_componants under the file timing_nearest_pair.h/.o
4. Lines 12 and 13 define the location of the .h and .o file for weight dependence component required by this model build. In this case it is located in plasticity/stdp/weight_dependence folder from neuron_componants under the fileweight_additive-one_term.impl.h/.o
5. Line 15 is the same as line 11 in **Code1**.

Discussion on how to add new plasticity is covered here.

To make a new Neuron model build, you must either:

1. Create a copy of the example builds discussed above,
2. Modify the names and component listings,
3. Mody Line 1 of the Makefile located in src->neuron->Makefile so that it includes your new build.

Or:

1. change the template's component listings directly.

Compiling a new model

Once the Makefile has been created, you can build the binary by simply typing:

```
make
```

As the build relies on header files that are not explicitly specified in the Makefile, some of the changes that you make may require you to clean the build before building it, by running

```
make clean
```

Finally, you can also build the application in debug mode by typing:

```
make DEBUG=DEBUG
```

This will enable the log_debug statements in the code, which print out information to the iobuf buffers on the SpiNNaker machine; to read this there is a simple application in the example folder that can read iobuf buffers, which can be invoked as follows:

```
python iobuf.py <machine_name> <app_name>
```

where `<machine_name>` is the hostname or IP address of the SpiNNaker board, and `<app_name>` is the name of application (without the .aplx extension). This will retrieve the iobuf buffers for every instance of the application that is found to be running.

C code file interfaces:

The rest of this section goes through the different components interfaces and tries to explain what each one does, for the case where you need to create a new component for your neuron build.

Models

The C code file defines the functions that make up the interface of the neuron API. Note that pointer types are automatically created for the data structures defined in **Code 3.**

```
neuron_t * → neuron_pointer_t
global_neuron_params_t * → global_neuron_params_pointer_t
```

Code3: pointer types for neuron_model.h

The neuron interface requires the following functions to be implemented:

- ```
  void neuron_model_set_global_neuron_params(
        global_neuron_params_pointer_t params)
  ```
  ```

This function is used to set the global parameters after they have been read by the initialization function. This would often be used to store the parameters in a static variable for later use.

- `state_t neuron_model_get_membrane_voltage(neuron_pointer_t neuron)`
  This function should return the membrane voltage of the neuron from the given neuron structure. This may simply return the value of a variable in the structure, or it might perform a more complex calculation to obtain the membrane voltage. The value returned is used for the recording of the membrane voltage in the simulation.

- `void neuron_model_print_parameters(restrict neuron_pointer_t neuron)`
  This function is only used when the neuron model is compiled in "debug" mode. It should use the "log_debug" function to print each of the state variables and parameters of the neuron that might be useful in debugging.

- `void neuron_model_print_state_variables(restrict neuron_pointer_t neuron)`

  This function is only used when the neuron model is compiled in "debug" mode. It should use the "log_debug" function to print each of the parameters of the neuron don't change during a run and that might be useful in debugging.

- `bool neuron_model_state_update(input_t exc_input, input_t inh_input, input_t external_bias, neuron_pointer_t neuron)`
  This function takes the excitatory and inhibitory input; any external bias input (used in some plasticity models); and a neuron data structure; and uses these to compute the new state of the given neuron at this timestep. This function is where any differential equation solving should be implemented. After the state update, the function should return whether the neuron is considered to have spiked as a boolean (true if the neuron has spiked, false otherwise). Note that the input does not specify current or conductance; no conversion of the weights are done before this function is called, other than any scaling performed in `neuron_model_convert_input`.

- `void neuron_model_has_spiked(neuron_pointer_t neuron);`
  This function is used to reset neuron parameters after it has spiked.

See neuron_model_my_model_curr_exp.c in the template for an example of an implementation of the neuron interface.

A number of other modules are available for use for performing mathematical functions as part of the neuron state update. The `spinn_common` library provides a number of efficient fixed-point implementations of common functions. This includes `random.h`, which provides random number generation, `normal.h`, which provides normal distributions, `exp.h`, which provides an exp function and `log.h` which provides a log function.

Input types

The C code file defines the functions that make up the interface of the input type API. The Input Type API requires the following interface functions to be implemented.

- ```
  static input_t input_type_get_input_value(input_t value,
                                            input_type_pointer_t input_type);
  ```

  This function supports the scaling of the input values into the neuron model.

- ```
  static input_t input_type_convert_excitatory_input_to_current(
                          input_t exc_input, input_type_pointer_t input_type,
                          state_t membrane_voltage);
  ```

  This function provides the ability to convert an excitatory input into a current based input (all our model internally operate on currents).

- ```
  static input_t input_type_convert_inhibitory_input_to_current(
                          input_t inh_input, input_type_pointer_t input_type,
                          state_t membrane_voltage);
  ```

  This function provides the ability to convert an inhibitory input into a current based input (all our model internally operate on currents).

Synapse types

The C code file defines the functions that make up the interface of the synapse type API. The synapse Type API requires the following interface functions to be implemented.

- ```
  static void synapse_types_shape_input(
                          input_t *input_buffers,
                          index_t neuron_index, synapse_param_t* parameters);
  ```

  Decays the stuff that's sitting in the input buffers (to compensate for the valve behaviour of a synapse

  in biology (spike goes in, synapse opens, then closes slowly)) as these have not yet been processed and applied to the neuron.

- ```
  static void synapse_types_add_neuron_input(
                          input_t *input_buffers, index_t synapse_type_index,
                          index_t neuron_index, synapse_param_t* parameters,
  ```
  ```

```
                                input_t input)
```

Adds the inputs for a given timer period to a given neuron that is being simulated by this model.

- `static input_t synapse_types_get_excitatory_input(`

  ```
                                  input_t *input_buffers, index_t neuron_index)
  ```

  Extracts the excitatory input buffers from the buffers available for a given neuron id.

- `static input_t synapse_types_get_inhibitory_input(`

  ```
                                  input_t *input_buffers, index_t neuron_index)
  ```

  Extracts the inhibitory input buffers from the buffers available for a given neuron id.

- `static const char *synapse_types_get_type_char(index_t synapse_type_index)`

  Returns a human readable character for the type of synapse. Examples would be X = excitatory types, I = inhibitory types etc etc.

- `static void synapse_types_print_parameters(synapse_param_t *parameters)`

  Prints the input for a neuron id given the available inputs currently only executed when the models are in debug mode, as the prints are controlled from the synapses.c _print_inputs method.

Threshold types

The C code file defines the functions that make up the interface of the threshold type API. The threshold Type API requires the following interface functions to be implemented.

- `static bool threshold_type_is_above_threshold(`

  ```
                                  state_t      value,      threshold_type_pointer_t
                      threshold_type)
  ```

  Determines if the value given is above the threshold value

Additional inputs

The C code file defines the functions that make up the interface of the additional input type API. The additional input Type API requires the following interface functions to be implemented.

- `static input_t additional_input_get_input_value_as_current(`

  ```
                                  additional_input_pointer_t additional_input,
  ```

```
                                    state_t membrane_voltage)
```

Gets the value of current provided by the additional input this timestep

- ```
  static void additional_input_has_spiked(
                                    additional_input_pointer_t additional_input)
  ```

Notifies the additional input type that the neuron has spiked.

Python PyNN Model

Once the C code has been constructed, the PyNN model must be created in Python to translate the PyNN parameters into a form that the C code can understand. In PyNN, populations can be made up of an arbitrary number of neurons, however to maintain real-time operation the number of neurons that are simulated on each core must be limited. The PACMAN module is used by sPyNNaker to partition the populations into subpopulations, based on the specified maximum number of atoms per core of the model, as well as the resources required by the synaptic matrix. The DataSpecification module is then used to write the data for each subpopulation. This is then loaded onto the machine, along with binary executable, using SpiNNMan.

As with the C code, there are number of components that can be re-used, so that only properties relevant to the new model itself need to be defined. This is done by constructing an individual component for:

1. Neuron model,
2. Input type,
3. Synapse type,
4. Threshold type,
5. Additional input.

These 5 components are then handed over to the main interface object that every neuron model has to extend.

If we look at the my_model_curr_exp in python_models -> builds directory, we will see the code shown in **Code 4** where the my_model_curr_exp builds its components and hands them over to the main sPyNNaker interface. The breakdown is as follows:

1. On Lines 106 to 108 the neuron model component is created.
2. On Lines 112 and 113 the synapse type component is created.
3. On Line 117 the input type component is created.
4. On Line 121 the threshold type component is created.
5. Line 125 shows that this model does not contain any additional input components.
6. Lines 129 to 141 show the handing over of these separate components to the sPyNNaker main system which will handle all the python support.

```
104 .    # TODO: create your neuron model class (change if required)
105.     # create your neuron model class
106.     neuron_model = NeuronModelLeakyIntegrateAndFire(
107.       n_neurons, machine_time_step, v_init, v_rest, tau_m, cm, i_offset,
108.       v_reset, tau_refrac)

110.     # TODO: create your synapse type model class (change if required)
```
```

```
111.    # create your synapse type model
112.    synapse_type = SynapseTypeExponential(
113.      n_neurons, machine_time_step, tau_syn_E, tau_syn_I)

115.    # TODO: create your input type model class (change if required)
116.    # create your input type model
117.    input_type = InputTypeCurrent()

119.    # TODO: create your threshold type model class (change if required)
120.    # create your threshold type model
121.    threshold_type = ThresholdTypeStatic(n_neurons, v_thresh)

123.    # TODO: create your own additional inputs (change if required).
124.    # create your own additional inputs
125.    additional_input = None

127.    # instantiate the sPyNNaker system by initializing
128.    #  the AbstractPopulationVertex
129.    AbstractPopulationVertex.__init__(
130.    # standard inputs, do not need to change.
131.      self, n_neurons=n_neurons, label=label,
132.      max_atoms_per_core=MyModelCurrExp._model_based_max_atoms_per_core,
133.      machine_time_step=machine_time_step,
134.      timescale_factor=timescale_factor,
135.      spikes_per_second=spikes_per_second,
136.      ring_buffer_sigma=ring_buffer_sigma,
137.      incoming_spike_buffer_size=incoming_spike_buffer_size,
138.      neuron_model=neuron_model, input_type=input_type,
139.      synapse_type=synapse_type, threshold_type=threshold_type,
140.      additional_input=additional_input,
141       model_name=self.model_name, binary=self._binary_name)
```

Code 4: Subsection of the my_model_curr_exp.py class

Take care to note that the same component names are used in the python as what are used in the c code's *Makefile*. This means for every new component you build for a neuron build in c which is not originally supported by the sPyNNaker tools, you need to build a corresponding python component file.

We have put in your new_template folder a set of template files within the template directory for each python component. These are located under python_models -> neuron_components.

Python __init__.py files

Most of the __init__.py files in the template do not contain any code. The one within `python_models` is the exception; this file adds the `model_binaries` module to the executable paths, allowing sPyNNaker to search this folder for your compiled binary. You can also import your module here to make it easy to use in other scripts.

Python setup.py file

This file enables you to install the new module. This is set up to install all the modules in the template; if you add any modules, these also need to be added to this file (it is not recursive; each module has to be added separately). To add the module to your python environment in such a way that you can still edit it, you can run:

```
[sudo] python setup.py develop [--user]
```

You need to use `sudo` if you are installing centrally on Linux or Mac OS X; on windows you need to be in an Administrative console. Add `--user` instead if you want to install only for your username (you shouldn't mix these two options, or you will end up installing it only for the root user).

Using your module

In order to use the new module, you need to import your module in addition to PyNN e.g. for the template module, you can do the following:

```
import pyNN.spiNNaker as p
import python_model as new_models
pop = p.Population(1, new_models.MyModelCurrExp)
```

A more detailed example is shown in the template in `examples/my_example.py`.

Task 1: A Simple Neural Model [Easy]

This task will create a simple neural model using the template, and execute it on SpiNNaker.

Change the template by adding two parameters, one representing a decay (default value of 0.1) and one representing a rest voltage (default value of -65.0). The parameters should be `REAL` values (`DataType.S1615`). Change the model to subtract the difference between the current voltage and the rest voltage multiplied by the decay from the membrane voltage, before adding the total input i.e.

```
v_membrane = v_membrane - ((v_membrane - v_rest) * decay) + input
```

Run the example script and see what happens.

Task 2: A Spiking Neuron Model [Moderate]

This task will look at adding a threshold at which the neuron spikes.

Add further parameters to the model created previously for the threshold voltage of the neuron (`REAL`, default value -60.0), the reset voltage (`REAL`, default value -70.0) and another parameter which is the refractory period (`uint32_t`, default value 2.0), in milliseconds. You will also need a state variable to keep a refractory timer (`int32_t`). Change the model C code to spike (return `true`) when the neuron voltage is greater than or equal to the threshold voltage after the update. If the neuron spikes, the voltage should then be set to the reset voltage, and the refractory timer should be set to the refractory period. Add a condition so that the neuron membrane voltage is only updated while the refractory timer is less than or equal to 0. If it is greater than 0, the refractory timer should be reduced by one.

Update the python code to match the C code. Note that the python code will need to convert the refractory period in milliseconds to the number of machine time steps.

Update the example script to record and plot the spikes, and run it again.

Task 3: A Stochastic Threshold Model [Hard]

This task will look at more complex model using some of the provided functions in the spinn_common library. Note that the models are automatically compiled with this library, so no additions to the Makefile are necessary.

Take the neuron model created in the previous task, and add a parameter representing the probability of the neuron firing if it is over the threshold value. This parameter will be between 0 and 1 in Python (default

of 0.5), but as the random number generator generates an integer value, this should be converted into a `uint32_t` value between 0 and 0x7FFFFFFF. Add a global parameter which is the seed of the random number generator. This is an array of 4 `uint32_t` values for the simplest random number generator in `normal.h`. Validate the Marsaglia KISS 64 RNG seed during initialisation of the global parameters (`validate_mars_kiss64_seed(mars_kiss64_seed_t seed)`). When the neural model is over the threshold voltage, call the RNG with the seed (`mars_kiss64_seed(mars_kiss64_seed_t seed)`). The neuron should only spike if the value is greater than than the probability.

Rerun the example script and see how the number of spikes differs for different settings of the spike probability.