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. [This is a draft for release version 5.0.0 of the software].

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:

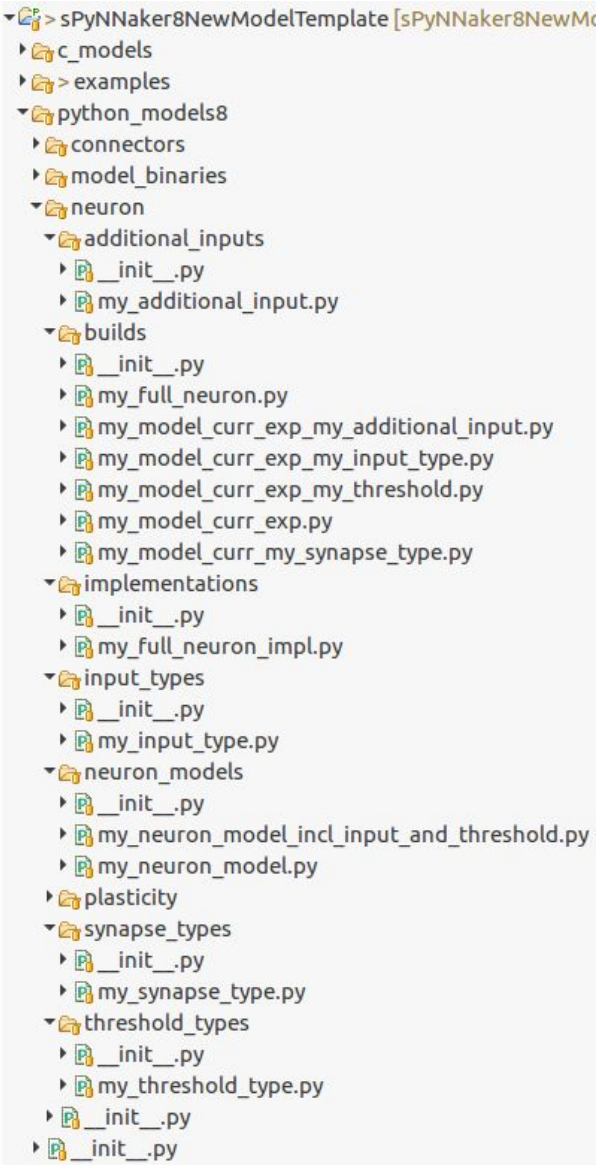
http://spinnakermanchester.github.io/spynnaker/5.0.0/PyNNOnSpiNNakerExtensions.html

Project Layout

The layout of the new model template API is shown below; this shows examples of models called "my_model_*", with various different user-defined components. It is recommended that the C and Python code is kept in the same project to help keep them synchronized.

C code

Python code



For SpiNNaker 5.0.0, this template structure can be downloaded from one of the following locations:
https://spinnakermanchester.github.io/latest/spynnaker_new_model_template_zip.html
https://spinnakermanchester.github.io/latest/spynnaker_new_model_template_tar_gz.html

If you have already set up a development environment for SpiNNaker in C and Python following the instructions at https://spinnakermanchester.github.io/development/devenv.html then this template structure is in its own repository at https://github.com/SpiNNakerManchester/sPyNNaker8NewModelTemplate.git.

C model builds

The standard implementation for neuron builds consists of a collection of components which when connected together produce a complete neural model. These components are defined in **Table 1**.

Component	Definition
Input type component	The type of input the model takes. Currently there are 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 type component	Determines the threshold of the membrane voltage which determines when the neuron spikes. Currently the only implementations are a static threshold and a stochastic Maass threshold.
Additional input type component	Any additional input current that might be based on the membrane voltage or other parameters.
Neuron model component	Determines how the neuron state changes over time and the outputs the current membrane voltage of the neuron. Currently there are Izhikevich (IZK) and Leaky-Integrate-and-Fire (LIF) implementations supported by sPyNNaker.
Synapse dynamics component	Determines how plasticity works within the model. sPyNNaker implements a model for no plasticity (i.e. static dynamics), and two different STDP dynamics models. Only static dynamics are considered in this tutorial.

Table 1: Different supported components

Each of these components can be used within an overall neuron implementation framework; alternatively, the user can define their own implementation framework that uses any (or none) of the above components as required. The standard implementation uses each of the components as described above.

Each build is stored within its own folder in the `c_models->src->my_models->builds` directory. Within each build is a *Makefile* containing the implementation framework and 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->makefiles->my_model_curr_exp->Makefile`

we can see the lines shown in **Code 1**.

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

```
2.
3.  NEURON_IMPL_H = $(SOURCE_DIR)/neuron/implementations/neuron_impl_standard.h
4.  NEURON_MODEL = $(EXTRA_SRC_DIR)/my_models/models/my_neuron_model_impl.c
5.  NEURON_MODEL_H = $(EXTRA_SRC_DIR)/my_models/models/my_neuron_model_impl.h
6.  INPUT_TYPE_H = $(SOURCE_DIR)/neuron/input_types/input_type_current.h
7.  THRESHOLD_TYPE_H = $(SOURCE_DIR)/neuron/threshold_types/threshold_type_static.h
8.  SYNAPSE_TYPE_H = $(SOURCE_DIR)/neuron/synapse_types/synapse_types_exponential_impl.h
9.  SYNAPSE_DYNAMICS = $(SOURCE_DIR)/neuron/plasticity/synapse_dynamics_static_impl.c
10.
11. include ../extra.mk
```

Code 1: my_model_curr_exp's Makefile

- Line 1 declares the name of the APP - here we are using the name of the current directory. (The aplx extension is added automatically.)
- Line 3 is the file that describes the implementation used in this neuron model to combine any (or none) of the separate components. This is implemented entirely in a header file.
- Lines 4 and 5 are the files that make up the neuron model component (described in **Table 1**) used for this model build (both the .c and .h files are needed). Note that these are stated to be in the $(EXTRA_SRC_DIR) folder - this is declared to be the c_models/src folder within the archive within Makefile.common. The sPyNNaker standard source files are declared to be within $(SOURCE_DIR), and these are used by other components.
- Line 6 states the input type component (described in **Table 1**) for this model. Input types are implemented entirely in a header file.
- Line 7 states the threshold type component (described in **Table 1**) for this model. Threshold types are implemented entirely in a header file.
- Line 8 states the synapse type component (described in **Table 1**) for this model. Synapse types are implemented entirely in a header file.
- Line 9 states the synapse dynamics component (described in **Table 1)** for this model.
- Line 11 tells the make system to import the Makefile `extra.mk` at the next level up so that it can detect where the rest of the code needed to be linked in here can be found.

Other Makefile instances might also include TIMING_DEPENDENCE_H and WEIGHT_DEPENDENCE_H - these are used when the synapse dynamics includes plasticity, and ADDITIONAL_INPUT_H for the additional_input_type component, where it is defined. A tutorial on how to add new plasticity implementations 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->my_models->Makefile` so that it includes your new build.

Or:

1. Change the template's component listings directly.

Compiling a new model

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 beforehand, by running

```
make clean
```

from the c_models directory. Once the Makefile has been edited appropriately, you can build the binary by simply changing to the directory containing the Makefile and typing:

```
make
```

It is also possible to build all the binaries defined in the template by running the `make` command from the top directory of the C code (c_models).

Assuming this builds with no errors, you should find the relevant aplx files have been built and placed into the python_models8->model_binaries directory (as specified by common.mk) in the Python code.

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

```
make SPYNNAKER_DEBUG=DEBUG
```

This will enable the log_debug statements in the code, which print out information to the iobuf buffers on the SpiNNaker machine (Note: be aware that this increases the instruction memory used by the code and may in some instances increase it beyond the 32kB limit). By default, the tools won't extract the printed error messages. To enable this behaviour, you can add the following to your **local** .spynnaker.cfg file:

```
[Reports]
extract_iobuf=True
```

The "iobuf" messages will then be downloaded after the execution is complete. These are stored relative to your executing script in reports/DATE-TIME-ID/run_N/provenance_data/ for appropriate values of DATE, TIME, ID and N; each is a .txt file containing any output printed from each core used during the execution of your program.

C code file interfaces

The first interface that is required is an implementation file; that is, basically, how each of the different components of the model are combined together at each time-step of the model. This implementation file requires the user to define five functions:

- `neuron_impl_initialise(uint32_t n_neurons)`

  In this function, the arrays to be used in the implementation need to be allocated within DTCM.

- `neuron_impl_add_inputs(index_t synapse_type_index, index_t neuron_index, input_t weights_this_timestep)`

  This function adds the inputs from the synapse into the neuron.

- `neuron_impl_load_neuron_parameters(address_t address, uint32_t next, uint32_t n_neurons)`

  This function copies parameters into DTCM from SDRAM.

- `neuron_impl_store_neuron_parameters(address_t address, uint32_t next, uint32_t n_neurons)`

  This function copies parameters from DTCM into SDRAM (e.g. to send back to the host).

- `neuron_impl_do_timestep_update(index_t neuron_index, input_t external_bias, state_t *recorded_variable_values)`

  This function describes what happens on the timestep update for this particular implementation.

The standard implementation using each of the components of the model can be found in the sPyNNaker installation in `neural_modelling->src->neuron->implementations->neuron_impl_standard.h` - use

this (as shown in the earlier Makefile) unless you wish to combine your components together in a non-standard manner.

Other examples of implementations can be found in the same directory in sPyNNaker, and the `src->my_models->implementations` directory in the new model template. These implementations use a different form of Makefile; see for example `c_models->makefiles->my_full_neuron_impl->Makefile`

```
1. APP = $(notdir $(CURDIR))
2.
3. NEURON_IMPL_H = $(EXTRA_SRC_DIR)/my_models/implementations/my_full_neuron_impl.h
4. SYNAPSE_DYNAMICS = $(SOURCE_DIR)/neuron/plasticity/synapse_dynamics_static_impl.c
5.
6. include ../extra.mk
```

Code 2: my_full_neuron_impl's Makefile

In this instance, the entirety of the model including all required components is defined in the implementation header file, so no component files need to be included apart from the SYNAPSE_DYNAMICS .c file. It is possible to write your own component header files and include them in the implementation header file without having to add anything further to this Makefile; however, if you use a .c file at all then you must add its full path to a variable called OTHER_SOURCES within this Makefile (see for an example the Makefile in sPyNNaker at `neural_modelling->makefiles->neuron->IF_curr_exp_sEMD->Makefile).`

The rest of this section goes through each interface for the different components of the neuron build and explains what each one does.

Neuron Models

The C header file defines:

- The neuron data structure `neuron_t`. This includes the parameters and state for each neuron to be executed on a core. This commonly includes the membrane voltage of the neuron, as well as an offset input current.
- The global parameters data structure `global_neuron_params_t`. This includes parameters that are shared across all neurons within a population. This might include such things as the time step of the simulation.

See `my_neuron_model_impl.h` in the template for an example of a header file. Comments show where the file should be updated to create your own model.

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 the header as follows:

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

- ```
  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_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 the value of the membrane voltage. Note that the input will always be presented as current - conductance input is converted to current input in the input type. Additionally, the input values are all positive, including the inhibitory input; thus if the total input current is being considered, the inhibitory input current should be subtracted from the excitatory input current.

- `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, and is taken before the state update is performed.

- `void neuron_model_has_spiked(neuron_pointer_t neuron);`
  This function is used to reset the parameters of the neuron after it has spiked. It is called only if the membrane voltage value returned from `neuron_model_state_update` is determined to be above the threshold determined by the threshold type.

- `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 state variables of the neuron that change during a run and that might be useful in debugging.

- `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 parameters of the neuron that don’t change during a run and that might be useful in debugging.

See `my_neuron_model_impl.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 mathematical and probabilistic functions. This includes `random.h`, which provides random number generation, `normal.h`, which provides normal distributions, `stdfix-exp.h`, which provides an exponential function and `log.h` which provides a logarithm function.

Synapse types

The synapse type header file defines the `synapse_param_t` data structure that determines the parameters required for shaping the synaptic input. For example, this might be done to compensate for the valve behaviour of a synapse in biology (spike goes in, synapse opens, then closes slowly). The parameters for all the synaptic inputs for a single neuron need to be defined in this structure; for example, if there are different parameters for excitatory and inhibitory neurons, both of these parameters must be explicitly defined in this structure. The structure might also contain parameters for computing the initial value that will be added to the input buffer following a spike from a preceding neuron.

Note that the input will have already been delayed by the appropriate amount before it reaches this function, and that the input weights from several spikes may be combined into a single weight. Additionally, the input weights might be either current or conductance as determined by the input type. The synapse type should not perform any conversion of the weights.

The synapse type header file also 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);`
  Shapes the values (current or conductance) in the input buffers for the synapses of a given neuron. The input buffers for all neurons and synapse types are given here, and the following function can be used to obtain the index of the appropriate input buffer given the indices of the neuron and of the synapse (e.g. if there is an excitatory and inhibitory synapse per neuron, the indices might be 0 and 1 respectively):
  `index_t synapse_types_get_input_buffer_index(synapse_index, neuron_index)`

- `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 a synaptic weight input to the input buffer for a given synapse of a given neuron after a spike has been received (and appropriately delayed). This allows the weight to be scaled as required before it is added to the buffer.

- `static input_t synapse_types_get_excitatory_input(`
  `input_t *input_buffers, index_t neuron_index)`
  Returns the total combined excitatory input from the buffers available for a given neuron id. Note that if several synapses are excitatory, this function should add up the input values (or perform an otherwise appropriate function) to return the total excitatory input value.

- `static input_t synapse_types_get_inhibitory_input(`
  `input_t *input_buffers, index_t neuron_index)`
  Extracts the total combined inhibitory input from the buffers available for a given neuron id. Note that if several synapses are inhibitory, this function should add up the input values (or perform an otherwise appropriate function) to return the total inhibitory input value. Note also that the value should be a positive number; subtraction is performed in the neuron model as required.

- `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 and I = inhibitory types.

- `static void synapse_types_print_input(`
  `input_t input_buffers, index_t neuron_index)`
  Prints the input for a neuron id given the available inputs. This is currently only executed when the models are in debug mode.

- `static void synapse_types_print_parameters(synapse_param_t *parameters)`
  Prints the static parameters of the synapse type. This is currently only executed when the models are in debug mode.

See `synapse_types_my_impl.h` for an example of an implementation of a synapse type.

Threshold types

The threshold type header file defines the `threshold_type_t` data structure that declares the parameters required for the threshold type. This might commonly include the actual threshold value amongst other parameters. The header also 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 threshold has been reached; if the neuron is to spike, given the value of the state variable, true is returned, otherwise false is returned.

See `my_threshold_type.h` for an example of an implementation of a threshold type.

Additional inputs

The additional input header file defines the `additional_input_t` data structure, which declares the parameters required for the additional input. The header also 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 may or may not be dependent on the membrane voltage.

- `static void additional_input_has_spiked(`
  `additional_input_pointer_t additional_input)`
  Notifies the additional input type that the neuron has spiked.

See `my_additional_input.h` for an example of an implementation of an additional input type. You will also need to add the line (e.g.)

```
ADDITIONAL_INPUT_H = $(EXTRA_SRC_DIR)/neuron/additional_inputs/my_additional_input.h
```

to the relevant Makefile when you wish to use these additional parameters in a model.

Python Model Builds

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 the 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 `my_model_curr_exp.py` in the `python_models8->neuron->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. A build such as this must inherit either from AbstractPyNNNeuronModel (this is the way to go if you have written a non-standard implementation), or AbstractPyNNNeuronModelStandard (when using a standard implementation). The breakdown is as follows:

1. On line 36 the neuron model component is created.
2. On lines 39 and 40 the synapse type component is created.
3. On line 43 the input type component is created.
4. On line 46 the threshold type component is created.
5. Additonal input type is optional, so in this instance it is not passed in.
6. Lines 49 to 59 show the handing over of these separate components to the model superclass, which, using the implementation described earlier, will pass the model to the sPyNNaker main system, and this will handle all the python support. Note that the binary must match the name of the aplx file generated by the C code.

```
35.     # create neuron model class
36.     neuron_model = MyNeuronModel(i_offset, my_neuron_parameter, v)
37.
38.     # create synapse type model
39.     synapse_type = SynapseTypeExponential(
40.       tau_syn_E, tau_syn_I, isyn_exc, isyn_inh)
41.
42.     # create input type model
43.     input_type = InputTypeCurrent()
44.
45.     # create threshold type model
46.     threshold_type = ThresholdTypeStatic(v_thresh)
47.
48.     # Create the model using the superclass
49.     super(MyModelCurrExp, self).__init__(
50.
51.       # the model name (shown in reports)
52.       model_name="MyModelCurrExp",
53.
54.       # the matching binary name
55.       binary="my_model_curr_exp.aplx",
56.
57.       # the various model types / components
58.       neuron_model=neuron_model, input_type=input_type
59.       synapse_type=synapse_type, threshold_type=threshold_type)
```

Code 4: Subsection of the my_model_curr_exp.py class

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

In the new_template folder there are a set of template files within the template directory for each python component. These are located under `python_models8->neuron`. These detail the parts of the class that can be changed for your model.

An example of how to incorporate a non-standard implementation is shown in the `my_full_neuron.py` in the `python_models8->neuron->builds` directory. In this instance, because all the required parameters are contained within the implementation, this simply requires the initialisation of an AbstractPyNNNeuronModel with the implementation MyFullNeuronImpl (see `python_models8->neuron->implementations->my_full_neuron_impl.py`) containing these particular parameters. If you have your own components to incorporate then these must be imported and passed in as necessary, so that the order of parameters sent to the machine is the same as the order of parameters as specified in the C code.

Python __init__.py files

Most of the __init__.py files in the template do not contain any code. The one within `python_models8` 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 created earlier, you can do the following:

```
import pyNN.spiNNaker as p
from python_models.neuron.builds.my_model_curr_exp import MyModelCurrExp
pop = p.Population(1, MyModelCurrExp, {})
```

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

Task 1: Simple Neuron Model [Easy]

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

1. Change the `my_neuron_model_impl.c` and `.h` templates by adding two parameters, one representing a decay and one representing a rest voltage. The parameters should be `REAL` values.
2. 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
```

3. Recompile the binary.
4. Update the python code model to accept the new decay and rest voltage parameters, ensuring that they match the order of the C code (use `DataType.S1615`). Add getters and setters for the values and update the number of neuron parameters.
5. Update the python code builds to accept the new parameters with default values of 0.1 for decay and -65.0 for the rest voltage. Don't forget to update the DataHolders as well!

Run the example script and see what happens.

Task 2: Conductance-based Model [Moderate]

This task will build a conductance-based model.

1. Make a copy of the C build folder for my_model_curr_exp to my_model_cond_exp.
2. Change the relevant Makefiles to use the conductance input type and ensure that the binary name is different from the current based model.
3. Build the binary.
4. Copy the python model `my_model_curr_exp.py` to `my_model_cond_exp.py` and update the code to use the conductance input type, including adding the new required parameters for conductance, and the binary name and model name.
5. Copy `my_model_curr_exp_data_holder.py` to `my_model_cond_exp_data_holder.py` and make sure the parameters are specified here too.
6. Update the example script to use the new model, adjusting the weights to be conductances (usually much smaller values e.g. 0.1 should be enough)

Run the example script and see what happens.

Task 3: Stochastic Threshold Model [Hard]

This task will create a new threshold model for stochastic thresholds.

1. Update the template threshold type `my_threshold_type.c` and `.h`, removing the parameter `my_threshold_parameter`, and adding a parameter representing the probability of the neuron firing if it is over the threshold value. This will be a `uint32_t` value in C (see later for details).

2. Add another 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 `random.h` (from the spinn_common library - as this should have been installed, you can use `#include <random.h>`).

3. Update the threshold calculation so that when the membrane voltage is over the threshold voltage, the RNG is called with the seed (`mars_kiss64_seed(mars_kiss64_seed_t seed)`).

4. Update the threshold calculation to only result in a spike if the value returned from the RNG is less than than the probability value.

5. Rebuild the C code.

6. Update the `my_threshold_type.py` python code to include the new parameters, and to generate the random seed. The probability 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. The seed can be generated using a NumpyRNG (from pyNN.random), which can be provided to the model as a parameter. Once generated, the seed should be validated using:

   ```
   spynnaker.pyNN.utilities.utility_calls.validate_mars_kiss_64_seed(seed)
   ```

   where `seed` is an array of 4 integer values. Note that `seed` will be updated in place.

7. Update the `my_model_curr_exp_my_threshold_type.py` build to include the new parameters and pass them in to the threshold type. Make rng an optional parameter, which if not set uses a new NumpyRNG.
8. Update the example script to decrease the threshold value to ensure that the model fires.

Run the example script and see how the number of spikes, and where these spikes occur, differs for different values of the spike probability.

Task 4: Write your own neuron implementation [Hard]

[Draft, to be updated] This task requires you to write your own neuron implementation.

1. A list of instructions here

2. Another instruction

3. And another one

4. And so on.

Run the example script and see ...