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.

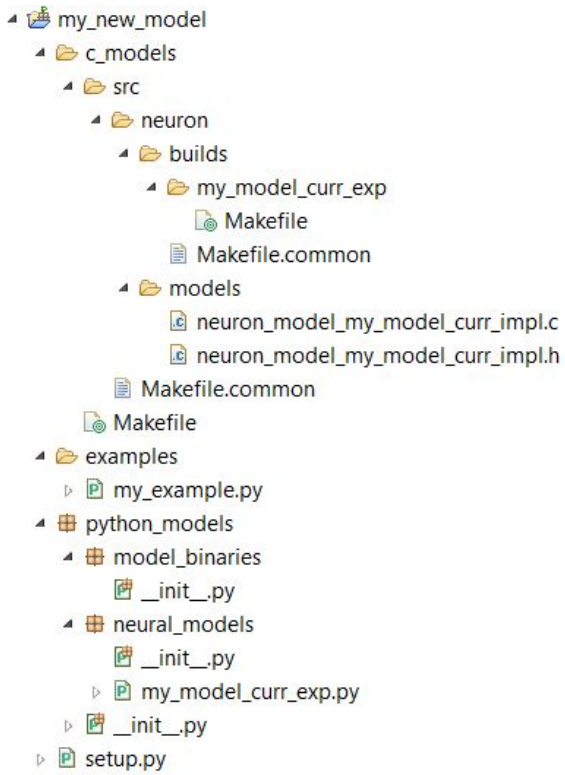
Installation

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/2015.005.Arbitrary/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.



This template structure can be downloaded from:
http://spinnakermanchester.github.io/2015.005.Arbitrary/template_new_model.zip

C code header file

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.

- A definition of a function `input_t neuron_model_convert_input(input_t input)`. This can be used to perform any scaling of input between the input buffers and the neuron model. This allows the values in the input buffers to maintain a higher level of precision for computation, but then returns the value to the expected scale for the input of the neuron. This is currently done for conductance-based models, where input conductances are usually in fractions of micro-siemens; in standard s16.15 fixed-point format, these fractional values waste a lot of the precision, since the top 16-bits will not be used. Multiplying every value by 1024 increases the resolution of these values, but the neuron model expects the values to be in microsiemens; this function divides the input values by 1024 when conductances are used (done using a right-shift by 10 for efficiency).

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

C code file

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
```

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(restrict neuron_pointer_t neuron)`
  This function is only used when the neuron model is compiled in “debug” mode (see later). 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.

- `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`.

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.

Makefiles

There are a number of Makefiles and Makefile.common files in the template. These have mostly been configured for you. The Makefile.common files set up directory structures, and include the appropriate Makefiles from the SpyNNaker module. The only Makefile that should need to be edited for your model is the one in the `build` folder for your executable; in the template, this is:

```
c_models/src/neuron/builds/my_model_curr_exp/Makefile
```

The Makefile builds the application that will run on SpiNNaker. This is made up of the neuron model that you have created, as well as the rest of the common neuron implementation. This includes the synapse dynamics that are supported by the build, as well as the synapse and spike processing code. The example provided is for a build with support for static synapses only. The inclusion of plastic synapses is covered elsewhere.

The important parts of this Makefile that need to be updated are:

- `MODEL_OBJS` - This includes a list of object files to be produced; note that the path of the object file is the same as the path of the C file, although the actual build process will push this into a build folder. In addition to the object file of the C file described above, this also includes the synapse dynamics type of the build (static in the template).
- `NEURON_MODEL_H` - This points at the neuron model header file.
- `SYNAPSE_TYPE_H` - This provides the synapse type for the build. 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).

Note that the combination of the `NEURON_MODEL_H` and `SYNAPSE_TYPE_H` will determine the overall model - thus `my_model_curr_exp` specifies the inclusion of `neuron_model_my_model_curr.h` as the model and `synapse_type_exponential_impl.h` as the synapse type.

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.

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 on to 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 extending the classes that define these properties. For example, the exponential synapse type described previously can be included by extending:

```
spynnaker.pyNN.models.abstract_models.abstract_model_components\
    .abstract_exp_population_vertex.AbstractExponentialPopulationVertex
```

To make use of the common codebase for all populations, all models must extend:

```
spynnaker.pyNN.models.abstract_models.abstract_population_vertex\
  .AbstractPopulationVertex
```

Once the class has been defined, a number of other properties and functions need to be defined:

- `_model_based_max_atoms_per_core` - This keeps track of the maximum neurons per core for this model. This is a user-configurable parameter, but the initial value is defined here as an absolute maximum. The absolute maximum supported by the data structures elsewhere in the C code is 255, so if you are unsure, you can use this value. If your model is particularly complex, you should set this to a lower value, as more processing time will be required per neuron.

- `__init__(self, n_neurons, machine_time_step, timescale_factor, spikes_per_second, ring_buffer_sigma, constraints=None, label=None, ...)`

  This is the initializer of the model class. The parameters of this initializer must include the following variables to match the sPyNNaker interface. These values will be passed down from sPyNNaker, and are mostly passed on to superclass initializers, to it is not critical that you understand what they all do here; however a short description is given below:
  - the number of neurons (`n_neurons`),
  - the machine timestep (`machine_time_step`) in microseconds,

- the timescale factor (`timescale_factor`),
- the maximum expected spikes per second (`spikes_per_second`),
- the number of sigmas in the ring buffer distribution to allow for (`ring_buffer_sigma`),
- any additional constraints on the vertex (`constraints`), and
- The vertex label (`label`)

In addition to these parameters, you should include any parameters required by the other classes that are extended. You can then also add your own parameters as required by your model. All the parameters must be given default values, including those required by the superclasses (not including those listed above, which are given values automatically); this allows the user to only specify those values that they want to change. The `machine_time_step` parameter can be used to convert any values that are in milliseconds to values that are in numbers of time steps (e.g. 10 milliseconds at a timestep of 100 microseconds is 10 / (100 / 1000.0) = 100 machine timesteps).

Within the init method, the following must be done:

- Call the initializer of `AbstractPopulationVertex`. This should pass on the parameters listed above, and should also pass the following parameters:
  - `n_params` - The total number of parameters and state variables defined in the `neuron_t` data structure in the C code. This may be more than the number of parameters passed to the initializer, as there may be variables that are not user-configurable.
  - `n_global_params` - The total number of parameters defined in the `neuron_global_params_t` data structure in the C code.
  - `binary` - This is the name of the executable that is produced when the Makefile runs. This is defined in the Makefile. This is the simple name of the file rather than the full path to the binary.
  - `weight_scale` - If any weight scaling is done in `neuron_model_convert_input`, the dividing factor must be specified here, so that the weights can be pre-multiplied by this amount. This value defaults to 1.0, so it doesn't need to be specified if it isn't used.
  - `max_atoms_per_core` - You can pass the `_model_based_max_atoms_per_core` variable defined above for this parameter. This will allow the user to configure this in a script if desired.
- Call the initializers of other included components. This will pass on the parameters required by these components.
- Store any of your own parameters. These are usually defined as:
  `self._<param> = utility_calls.convert_param_to_numpy(<param>, n_neurons)`
  The user can specify parameters using a single value; a list of values (with one per neuron); or a `RandomDistribution` object. The `convert_param_to_numpy` function normalises these into a numpy array.

- `initialize_<var>` - This is a set of functions, one for each of the state variables (i.e. those that change during the simulation). This allows the user to call the PyNN initialize() function. Note that the initial values are usually also available via the class constructor (usually postfixed with _init), but that PyNN doesn't usually expose state variables in this way. These functions should also make

use of the `utility_calls.convert_param_to_numpy` function to normalise the parameter values.

- Property and setter for each parameter (i.e. those variables that don't change during the simulation). This allows the user to call the PyNN set() function for these variables. As with the above the parameter values should be normalised.

- `model_name(self)` - This property (use the `@property` decorator) simply returns the model name.

- `set_model_max_atoms_per_core(new_value)` - This static method (use the `@staticmethod` decorator) sets the value of `_model_based_max_atoms_per_core`, to allow the user to override the default (with a smaller value).

- `get_cpu_usage_for_atoms(self, vertex_slice, graph)` - This method returns an estimate of the number of cpu clock cycles required per timer tick to run the model for the number of neurons given by `vertex_slice.n_atoms`. This value is unlikely to be critical at this point, so using the function as it is specified in the template should work correctly.

- `get_parameters(self)` - This method returns an array of `NeuronParameter` instances to match the parameters and state variables defined in the `neuron_t` data structure in the C header file. The order of the array must match the order of the parameters and state variables as defined in the `neuron_t` data structure. Along with each parameter, the data type must also be given. This is done using the `DataType` enum; the most commonly used values are `DataType.S1615`, which corresponds to a C fixed-point `accum` data type (or `REAL` in most of our code); and `DataType.UINT32` and `DataType.INT32` which correspond to unsigned and signed 32-bit integers respectively (`uint32_t` and `int32_t`).

- `get_global_parameters(self)` - This method returns an array of `NeuronParameter` instances. This must return the values of the global parameters exactly as defined in the `global_neuron_params_t` data structure in the C header file, including the order of the parameters.

- `is_population_vertex(self)` - This method needs to be defined, but just needs to return `True`. This is used to keep track of the types of the vertices.

- Any additional abstract methods defined by the extended classes. For example, `AbstractExponentialPopulationVertex` requires the `is_exponential(self)` method to return `True`.

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.