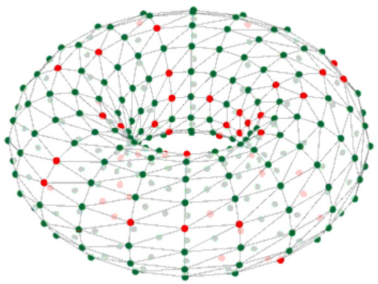
Adding New Neuron Models



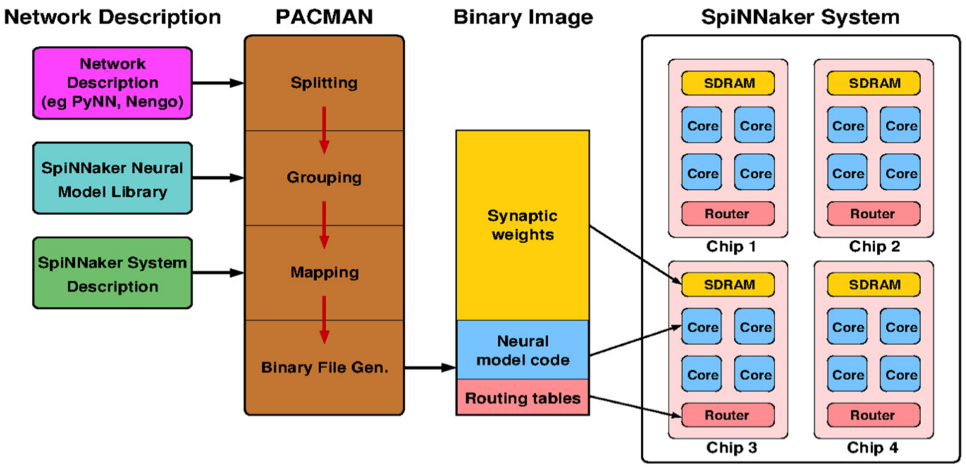
Andrew Rowley, Michael Hopkins

SpiNNaker Workshop, September 2016

Required code separation

- Any new neuron model requires both C and Python code
- C code makes the actual executable (on SpiNNaker), Python code configures the setup and load phases (on the host)
- These are separate but <u>must be</u> perfectly coordinated
- In almost all cases, the C code will be solving an ODE which describes how the neuron state evolves over time and in response to input

Required code separation



We will first describe the C requirements...

C Data Structures and Parameters

- The parameters and state of a neuron at any point in time need to be stored in memory
- For each neuron, the C header defines the ordering and size of each stored value
- The C types can be standard integer and floating-point, or ISO draft standard fixed-point, as required (see later talk *Maths & fixed-point libraries*)
- There is also one global data structure which services all neurons on a core

So here is an example using the Izhikevich neuron...

Specific neuron model – data structure

```
#include "neuron-model.h"

//  Izhikevic neuron data structure defined in neuron_model_izh_curr_impl.h

typedef struct neuron_t {

// 'fixed' parameters - abstract units
   REAL    A;
   REAL    B;
   REAL    C;
   REAL    D;

// variable-state parameters
   REAL    V;          // nominally in [mV]
   REAL    U;

// offset current [nA]
   REAL    I_offset;

// private variable used internally in C code
   REAL    this_h;

} neuron_t;

...
```

Global data structure

```
...

/*
   Global data structure defined in neuron_model_izh_curr_impl.h
*/

typedef struct global_neuron_params_t {

// Machine time step in milliseconds
   REAL    machine_timestep_ms;

} global_neuron_params_t;
```

Implementing the state update

- Neuron models are typically described as systems of initial value ODEs
- At each time step, the internal state of each neuron needs to be updated in response to inherent dynamics and synaptic input
- There are many ways to achieve this; there will usually be a 'best approach' (in terms of balance between accuracy & efficiency) for each neuron model
- A recently published paper gives a lot more detail: Hopkins & Furber (2015), "Accuracy and Efficiency in Fixed-Point Neural ODE Solvers" , *Neural Computation* **27**, 1–35
- The key function will always be *neuron_model_state_update*(); the other functions are mainly to support this and allow debugging etc.

Continuing the example by describing the key interfaces...

Neuron model API

```
// pointer to a neuron data type - used in all access operations
typedef struct neuron_t* neuron_pointer_t;

// set the global neuron parameters
void neuron_model_set_global_neuron_params( global_neuron_params_pointer_t params );

// key function in timer loop that updates neuron state and returns membrane voltage
state_t neuron_model_state_update(
            input_t exc_input, input_t inh_input, input_t external_bias,
            neuron_pointer_t neuron );

// return membrane voltage (= first state variable) for a given neuron
state_t neuron_model_get_membrane_voltage( restrict neuron_pointer_t neuron );

// update the neuron structure to take account of a spike
void neuron_model_has_spiked( neuron_pointer_t neuron );

// print out neuron definition and/or state variables (for debug)
void neuron_model_print_parameters( restrict neuron_pointer_t neuron );
void neuron_model_print_state_variables( restrict neuron_pointer_t neuron );
```

Specific neuron model – key functions

```
/* simplified version of Izhikevic neuron code defined in neuron_model_izh_curr_impl.c  */


// key function in timer loop that updates neuron state and returns membrane voltage
state_t neuron_model_state_update(
            input_t exc_input, input_t inh_input, input_t external_bias,
            neuron_pointer_t neuron ) {

// collect inputs
    input_t input_this_timestep =
        exc_input - inh_input + external_bias + neuron->I_offset;

// most balanced ESR update found so far
    _rk2_kernel_midpoint( neuron->this_h, neuron, input_this_timestep );
    neuron->this_h = global_params->machine_timestep_ms;

// return the value of the membrane voltage
    return neuron->V;
}


// make the discrete changes to state after a spike has occurred
void neuron_model_has_spiked( neuron_pointer_t neuron ) {
    neuron->V  = neuron->C;        // reset membrane voltage
    neuron->U += neuron->D;        // offset 2nd state variable
}
```

Makefile

```
APP = my_model_curr_exp

# This is the folder where things will be built (this will be created)
BUILD_DIR = build/

# This is the neuron model implementation
NEURON_MODEL = $(EXTRA_SRC_DIR)/neuron/models/my_neuron_model_impl.c

# This is the header of the neuron model, containing the definition of neuron_t
NEURON_MODEL_H = $(EXTRA_SRC_DIR)/neuron/models/neuron_model_my_model_curr_exp.h

# This is the header containing the input type (current in this case)
INPUT_TYPE_H = $(SOURCE_DIR)/neuron/input_types/input_type_current.h

# This is the header containing the threshold type (static in this case)
THRESHOLD_TYPE_TYPE_H = $(SOURCE_DIR)/neuron/threshold_types/threshold_type_static.h

# This is the header containing the synapse shaping type (exponential in this case)
SYNAPSE_TYPE_H = $(SOURCE_DIR)/neuron/synapse_types/synapse_types_exponential_impl.h

# This is the synapse dynamics type (in this case static i.e. no synapse dynamics)
SYNAPSE_DYNAMICS = $(SOURCE_DIR)/neuron/plasticity/synapse_dynamics_static_impl.c

# This includes the common Makefile that hides away the details of the build
include ../Makefile.common
```


Threshold Models – Interface and Implementation

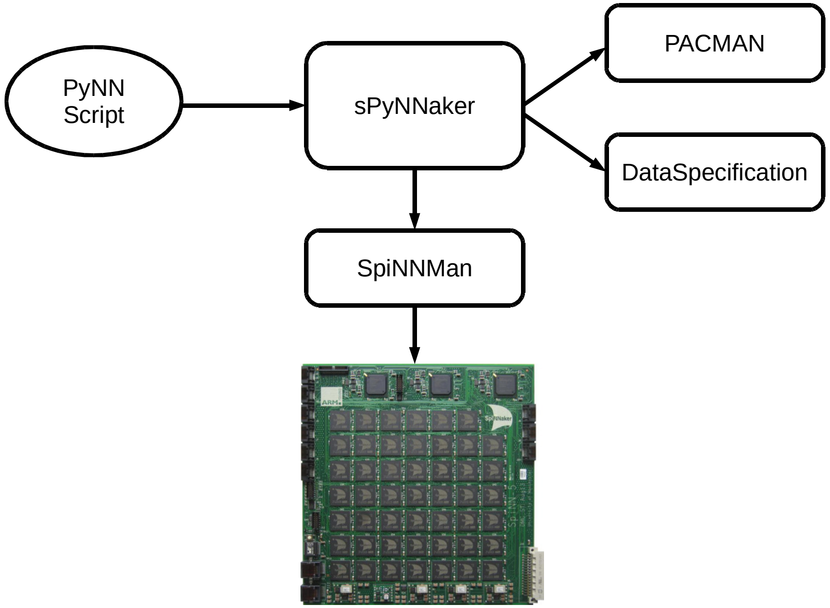
Interface

```
// Pointer to threshold data type - used to access all operations
typedef struct threshold_type_t;


// Main interface function - determine if the value is above the threshold
static inline bool threshold_type_is_above_threshold(
        state_t value, threshold_type_pointer_t threshold_type );
```

Static Threshold Implementation

```
typedef struct threshold_type_t {

    // The value of the static threshold
    REAL threshold_value;

} threshold_type_t;


static inline bool threshold_type_is_above_threshold(
        state_t value, threshold_type_pointer_t threshold_type ) {

    return REAL_COMPARE( value, >=, threshold_type•threshold_value );
}
```

Python Interface – Why?



MANCHESTER 1824

Python Interface

```
from spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model \
    import AbstractNeuronModel

class NeuronModelIzh(AbstractNeuronModel):
    def __init__(self, n_neurons, a, b, c, d, v_init, u_init, i_offset):
        AbstractNeuronModel.__init__(self)
        self._n_neurons = n_neurons
```


MANCHESTER 1824

Python Interface - Parameters

- Parameters can be:
  - Individual values
  - Array of values (one per neuron)
  - RandomDistribution
- Normalise Parameters
  - utility_calls.convert_param_to_numpy(
    param, n_neurons)

MANCHESTER 1824

Python Interface – initializer

```
from spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model \
    import AbstractNeuronModel

class NeuronModelIzh(AbstractNeuronModel):
    def __init__(self, n_neurons, a, b, c, d, v_init, u_init, i_offset):
        AbstractNeuronModel.__init__(self)
        self._n_neurons = n_neurons

        self._a = utility_calls.convert_param_to_numpy(a, n_neurons)
        self._b = utility_calls.convert_param_to_numpy(b, n_neurons)
        self._c = utility_calls.convert_param_to_numpy(c, n_neurons)
        self._d = utility_calls.convert_param_to_numpy(d, n_neurons)
        self._v_init = utility_calls.convert_param_to_numpy(v_init, n_neurons)
        self._u_init = utility_calls.convert_param_to_numpy(u_init, n_neurons)
        self._i_offset = utility_calls.convert_param_to_numpy(
            i_offset, n_neurons)
```


MANCHESTER 1824

Python Interface – properties

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    @property
    def a(self):
        return self._a

    @a.setter
    def a(self, a):
        self._a = utility_calls.convert_param_to_numpy(a, self.n_atoms)

    @property
    def b(self):
        return self._b

    @b.setter
    def b(self, b):
        self._b = utility_calls.convert_param_to_numpy(b, self.n_atoms)

    ...
```

Python Interface – state initializers

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def initialize_v(self, v_init):
        self._v_init = utility_calls.convert_param_to_numpy(v_init, self.n_atoms)

    def initialize_u(self, u_init):
        self._u_init = utility_calls.convert_param_to_numpy(u_init, self.n_atoms)
```


Python Interface – parameters

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def get_n_neural_parameters(self):
        Return 8

    def get_parameters(self):
        return [
            # REAL a
            NeuronParameter(self._a, DataType.S1615),
            # REAL b
            NeuronParameter(self._b, DataType.S1615),
            # REAL c
            NeuronParameter(self._c, DataType.S1615),
            # REAL d
            NeuronParameter(self._d, DataType.S1615),
            # REAL v
            NeuronParameter(self._v_init, DataType.S1615),
            # REAL u
            NeuronParameter(self._u_init, DataType.S1615),
            # REAL I_offset
            NeuronParameter(self._i_offset, DataType.S1615),
            # REAL this_h
            NeuronParameter(self._machine_time_step / 1000.0, DataType.S1615)
        ]
```


Python Interface – global params

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def get_n_global_parameters(self):
        return 1

    @inject_items({"machine_time_step": "MachineTimeStep"})
    def get_global_parameters(self, machine_time_step):
        return [
            NeuronParameter(machine_time_step / 1000.0, DataType.S1615)
        ]
```


Python Interface - Injection

```
@inject_items({"machine_time_step": "MachineTimeStep"})
def get_global_parameters(self, machine_time_step):
```

- Some items can be “injected” from the interface
  - Specify a dictionary of parameter name to “type” to inject
  - Parameter is in addition to the interface
- Common types include:
  - MachineTimeStep
  - TimeScaleFactor
  - TotalRunTime

Python Interface – CPU usage

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def get_n_cpu_cycles_per_neuron(self):

        # A bit of a guess
        return 150
```


Python Interface – Threshold type

```
class ThresholdTypeStatic(AbstractThresholdType):
    """ A threshold that is a static value
    """

    def __init__(self, n_neurons, v_thresh):
        AbstractThresholdType.__init__(self)
        self._n_neurons = n_neurons
        self._v_thresh = utility_calls.convert_param_to_numpy(
            v_thresh, n_neurons)
```

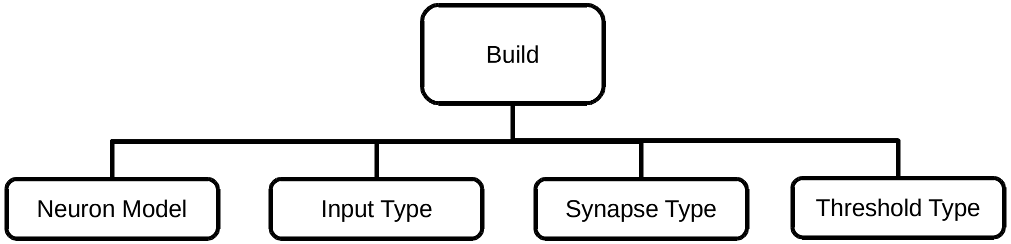

Python Interface – Threshold type

```
class ThresholdTypeStatic(AbstractThresholdType):
    """ A threshold that is a static value
    """
    ...

    @property
    def v_thresh(self):
        return self._v_thresh

    @v_thresh.setter
    def v_thresh(self, v_thresh):
        self._v_thresh = utility_calls.convert_param_to_numpy(
            v_thresh, self._n_neurons)
```


Python Interface – Threshold type

```
class ThresholdTypeStatic(AbstractThresholdType):
    """ A threshold that is a static value
    """
    ...

    def get_n_threshold_parameters(self):
        return 1

    def get_threshold_parameters(self):
        return [
            NeuronParameter(self._v_thresh, DataType.S1615)
        ]

    def get_n_cpu_cycles_per_neuron(self):

        # Just a comparison, but 2 just in case!
        return 2
```

Python Build



Python Build – Class Definition

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

class IzkCurrExp(AbstractPopulationVertex):
```


Python Build

```
class IzkCurrExp(AbstractPopulationVertex):
    _model_based_max_atoms_per_core = 255
    default_parameters = {
        'a': 0.02, 'c': -65.0, 'b': 0.2, 'd': 2.0, 'i_offset': 0,
        'u_init': -14.0, 'v_init': -70.0, 'tau_syn_E': 5.0, 'tau_syn_I': 5.0}
```


Python Build – initializer

```
class IzkCurrExp(AbstractPopulationVertex):
    def __init__(
            self, n_neurons, spikes_per_second=None, ring_buffer_sigma=None,
            incoming_spike_buffer_size=None, constraints=None, label=None,
            a=default_parameters['a'], b=default_parameters['b'],
            c=default_parameters['c'], d=default_parameters['d'],
            i_offset=default_parameters['i_offset'],
            u_init=default_parameters['u_init'],
            v_init=default_parameters['v_init'],
            tau_syn_E=default_parameters['tau_syn_E'],
            tau_syn_I=default_parameters['tau_syn_I']):

        neuron_model = NeuronModelIzh(
            n_neurons, a, b, c, d, v_init, u_init, i_offset)
        synapse_type = SynapseTypeExponential(
            n_neurons, tau_syn_E, tau_syn_I)
        input_type = InputTypeCurrent()
        threshold_type = ThresholdTypeStatic(n_neurons, _IZK_THRESHOLD)

        AbstractPopulationVertex.__init__(
            self, n_neurons=n_neurons, binary="IZK_curr_exp.aplx", label=label,
            max_atoms_per_core=IzkCurrExp._model_based_max_atoms_per_core,
            spikes_per_second=spikes_per_second,
            ring_buffer_sigma=ring_buffer_sigma,
            incoming_spike_buffer_size=incoming_spike_buffer_size,
            model_name="IZK_curr_exp", neuron_model=neuron_model,
            input_type=input_type, synapse_type=synapse_type,
            threshold_type=threshold_type, constraints=constraints)
```

Python Build – max atoms

```python
class IzkCurrExp(AbstractPopulationVertex):

    ...

    @staticmethod
    def set_model_max_atoms_per_core(new_value):
        IzhikevichCurrentExponentialPopulation.\
            _model_based_max_atoms_per_core = new_value

    @staticmethod
    def get_max_atoms_per_core():
        return IzkCurrExp._model_based_max_atoms_per_core
```


New Model Template

```
c_models
  src
    neuron
      additional_inputs
        my_additional_input.h
      builds
        my_model_curr_exp
          build
          Makefile
        my_model_curr_exp_my_additional_input
        my_model_curr_exp_my_threshold
        my_model_curr_exp_stdp_mad_my_timing_my_weight
        my_model_curr_my_synapse_type
        Makefile.common
      models
        my_neuron_model_impl.c
        my_neuron_model_impl.h
      plasticity
      synapse_types
        synapse_types_my_impl.h
      threshold_types
        my_threshold_type.h
      Makefile
    Makefile.common
  Makefile
```

```
examples
  __init__.py
  my_example.py
python_models
  connectors
  model_binaries
  neuron
    additional_inputs
      __init__.py
      my_additional_input.py
    builds
      __init__.py
      my_model_curr_exp_my_additional_input.py
      my_model_curr_exp_my_threshold.py
      my_model_curr_exp.py
      my_model_curr_my_synapse_type.py
    neuron_models
      __init__.py
      my_neuron_model.py
    plasticity
      stdp
      __init__.py
    synapse_types
      __init__.py
      my_synapse_type.py
    threshold_types
      __init__.py
      my_threshold_type.py
    __init__.py
```


Using Your Model

```python
import pyNN.spiNNaker as p
import python_models as new_models

my_model_pop = p.Population(
    1, new_models.MyModelCurrExp,
    {"my_parameter": 2.0,
     "i_offset": i_offset},
    label="my_model_pop")
```