Adding new models of synaptic plasticity

Oliver Rhodes, Andrew Rowley, Andrew Gait

SpiNNaker Workshop
October 2017
Overview

- Introduction to STDP
- Using STDP via PyNN
- Implementing STDP rules via sPyNNaker
What is STDP?

“Cells that fire together, wire together”
Spike Time Dependent Plasticity: Potentiation

\[ w + \Delta w \]

1

2

\[ \Delta t \]

\[ \Delta w \]
Spike Time Dependent Plasticity: Depression

\[ w - \Delta w \]

\[ \Delta w \]

\[ \Delta t \]
STDP Rules

\[
A_{\text{plus}} = e^{-\frac{\Delta t}{\tau_{\text{plus}}}}
\]

\[
A_{\text{minus}} = -e^{-\frac{\Delta t}{\tau_{\text{minus}}}}
\]
Simulating STDP - Weight update

Pre-synaptic weight update

\[ \Delta w_{ij}^- = F_-(w_{ij}) y_i(t^f_j) \]

Post-synaptic weight update

\[ \Delta w_{ii}^+ = F_+(w_{ii}) x_j(t^f_i) \]
STDP in PyNN
import spynnaker8 as sim
import pyNN.utility.plotting as plot
import matplotlib.pyplot as plt
n_neurons = 100
simtime = 5000
sim.setup(timestep=1.0)

pre_pop = sim.Population(n_neurons, sim.IF_curr_exp(), label="Pre")
post_pop = sim.Population(n_neurons, sim.IF_curr_exp(), label="Post")
pre_noise = sim.Population(n_neurons, sim.SpikeSourcePoisson(rate=10.0), label="Noise_Pre")
post_noise = sim.Population(n_neurons, sim.SpikeSourcePoisson(rate=10.0),
                            label="Noise_Post")

pre_pop.record("spikes")
post_pop.record("spikes")
STDP in PyNN

training = sim.Population(n_neurons,
    sim.SpikeSourcePoisson(rate=10.0, start=2000.0, duration=1000.0),
    label="Training")

sim.Projection(pre_noise, pre_pop, sim.OneToOneConnector(),
    synapse_type=sim.StaticSynapse(weight=2.0))

sim.Projection(post_noise, post_pop, sim.OneToOneConnector(),
    synapse_type=sim.StaticSynapse(weight=2.0))

sim.Projection(training, pre_pop, sim.OneToOneConnector(),
    synapse_type=sim.StaticSynapse(weight=5.0, delay=1.0))

sim.Projection(training, post_pop, sim.OneToOneConnector(),
    synapse_type=sim.StaticSynapse(weight=5.0, delay=10.0))
STDP in PyNN

timing_rule = sim.SpikePairRule(tau_plus=20.0, tau_minus=20.0, A_plus=0.5, A_minus=0.5)

weight_rule = sim.AdditiveWeightDependence(w_max=5.0, w_min=0.0)

stdp_model = sim.STDPMechanism(timing_dependence=timing_rule, weight_dependence=weight_rule, weight=0.0, delay=5.0)

stdp_projection = sim.Projection(pre_pop, post_pop, sim.OneToOneConnector(), synapse_type=stdp_model)
STDP: PyNN 0.75 vs PyNN 0.8

**PyNN 0.75**

```python
p.Projection(
    p1, p2, ..., synapse_dynamics=p.SynapseDynamics(
        slow=p.STDPMechanism(
            timing_dependence=sim.SpikePairRule(tau_plus=16.7, tau_minus=33.7),
            weight_dependence=sim.AdditiveWeightDependence(
                w_min=0.0, w_max=1.0, A_plus=0.005, A_minus=0.005))))
```

**PyNN 0.8**

```python
p.Projection(
    p1, p2, ..., synapse_type=p.STDPMechanism(
        timing_dependence=sim.SpikePairRule(
            tau_plus=20, tau_minus=20.0, A_plus=0.005, A_minus=0.005),
            weight_dependence=sim.AdditiveWeightDependence(w_min=0, w_max=1.0),
            weight=0.1, delay=2))
```
STDP in PyNN

sim.run(simtime)

pre_neo = pre_pop.get_data(variables="spikes")
pre_spikes = pre_neo.segments[0].spiketrains

post_neo = post_pop.get_data(variables="spikes")
post_spikes = post_neo.segments[0].spiketrains

print stdp_projection.getWeights()

sim.end()
STDP in PyNN

```python
line_properties = [{'color': 'red'}, {'color': 'blue'}]
plot.Figure(
    # plot spikes
    plot.Panel(pre_spikes, post_spikes, yticks=True,
               markersize=5, xlim=(0, simtime),
               line_properties=line_properties),
    plot.Panel(pre_spikes, yticks=True, markersize=5,
               xlim=(0, simtime), color='red',
               data_labels=['pre']),
    plot.Panel(post_spikes, yticks=True,
               markersize=5,
               xlim=(0, simtime), color='blue',
               data_labels=['post']),
    title="Balanced Random Network Example",
    annotations="Simulated with{}".format(sim.name())
)
plt.show()
```
STDP in PyNN

```python
line_properties = [{'color': 'red', 'markersize': 5}, {'color': 'blue', 'markersize': 2}]

plot.Figure(
    # plot spikes
    plot.Panel(pre_spikes, post_spikes, yticks=True, xlim=(0, simtime), line_properties=line_properties),
    title="STDP Network Example",
    annotations="Simulated with {}".format(sim.name()))
plt.show()
```
Software Stack

**User Interface (Host-side)**
- Front end interface (PyNN, graph, etc.)
- Mapping
- Python interface to SpiNNaker hardware
  - Ethernet if

**SpiNNaker Node**
- User Application Code
  - Application code e.g. LIF neurons, spike server, finite element analysis
  - Support for application event management and synchronization
  - Event-driven API (optional)
  - SARK
  - System management software
  - SpiNNaker hardware
  - Ethernet if
STDP API

Python (Host)

C (SpiNNaker)

[Diagram of Python and C environments related to STDP API]
sPyNNaker STDP Framework

Update synaptic weights based on relative spike timing

- **Timing rule**
  - Define relationship between spike times

- **Weight rule**
  - Define weight update based on timing data

- **Synapse Dynamics**
  - Define pre- and post-synaptic spike event handling

- **Synapse Structure**
  - Define plastic weight data structure
Timing Dependence Rule

Define how relative spike times are interpreted

Python:
- Provide user interface via PyNN
- Initialise and write rule parameters for loading to machine
- Extend AbstractTimingDependence

C:
- Compile into application binary
- Implement timing rule and runtime data-handling
- Follow timing_dependence.c/h
Extend AbstractTimingDependence
● Constructor to initialise objects and parameters
● Define rule parameters as properties

class TimingDependenceSpikePair(AbstractTimingDependence):

def __init__(self, tau_plus=20.0, tau_minus=20.0):
    AbstractTimingDependence.__init__(self)
    self._tau_plus = tau_plus
    self._tau_minus = tau_minus

    self._synapse_structure = SynapseStructureWeightOnly()

    # provenance data
    self._tau_plus_last_entry = None
    self._tau_minus_last_entry = None
Define properties:

- Rule parameters
- Instruct toolchain which binary to load

```python
@property
def tau_plus(self):
    return self._tau_plus

@property
def tau_minus(self):
    return self._tau_minu

@property
def vertex_executable_suffix(self):
    return "pair"
```
Manage memory

- Define pre trace bytes and weight terms
- Define overall parameter memory

```python
@property
def pre_trace_n_bytes(self):
    return 2

@overrides(AbstractTimingDependence.get_parameters_sdram_usage_in_bytes)
def get_parameters_sdram_usage_in_bytes(self):
    return 2 * (LOOKUP_TAU_PLUS_SIZE + LOOKUP_TAU_MINUS_SIZE)

@property
def n_weight_terms(self):
    return 1
```
Write rule parameters

- Spike pair rule uses lookup tables for performance
- Helper method to aid creation in correct format

```python
def write_parameters(self, spec, machine_time_step, weight_scales):
    # Check timestep is valid
    if machine_time_step != 1000:
        raise NotImplementedError(
            "STDP LUT generation currently only supports 1ms timesteps")

    # Write lookup tables
    self._tau_plus_last_entry = plasticity_helpers.write_exp_lut(
        spec, self._tau_plus, LOOKUP_TAU_PLUS_SIZE,
        LOOKUP_TAU_PLUS_SHIFT)
    self._tau_minus_last_entry = plasticity_helpers.write_exp_lut(
        spec, self._tau_minus, LOOKUP_TAU_MINUS_SIZE,
        LOOKUP_TAU_MINUS_SHIFT)
```
Initialise rule data from SDRAM

address_t timing_initialise(address_t address) {

    log_info("timing_initialise: starting");
    log_info("\tSTDP pair rule");

    // Copy LUTs from following memory
    address_t lut_address = maths_copy_int16_lut(&address[0], TAU_PLUS_SIZE,
                                                &tau_plus_lookup[0]);

    lut_address = maths_copy_int16_lut(lut_address, TAU_MINUS_SIZE,
                                       &tau_minus_lookup[0]);

    log_info("timing_initialise: completed successfully");

    return lut_address;
}
```c
static inline pre_trace_t timing_add_pre_spike(
    uint32_t time, uint32_t last_time, pre_trace_t last_trace) {

    // Get time since last spike
    uint32_t delta_time = time - last_time;

    // Decay previous r1 and r2 traces
    int32_t decayed_r1_trace = STDP_FIXED_MUL_16X16(
        last_trace, DECAY_LOOKUP_TAU_PLUS(delta_time));

    // Add energy caused by new spike to trace
    int32_t new_r1_trace = decayed_r1_trace + STDP_FIXED_POINT_ONE;

    log_debug("\tdelta_time=%u, r1=%d\n", delta_time, new_r1_trace);

    // Return new pre-synaptic event with decayed trace values with energy
    // for new spike added
    return (pre_trace_t) new_r1_trace;
}
```
static inline post_trace_t timing_add_post_spike(
    uint32_t time, uint32_t last_time, post_trace_t last_trace) {

    // Get time since last spike
    uint32_t delta_time = time - last_time;

    // Decay previous o1 and o2 traces
    int32_t decayed_o1_trace = STDP_FIXED_MUL_16X16(last_trace,
            DECAY_LOOKUP_TAU_MINUS(delta_time));

    // Add energy caused by new spike to trace
    int32_t new_o1_trace = decayed_o1_trace + STDP_FIXED_POINT_ONE;

    log_debug("\tdelta_time=%d, o1=%d\n", delta_time, new_o1_trace);

    // Return new pre-synaptic event with decayed trace values with energy
    // for new spike added
    return (post_trace_t) new_o1_trace;
}
```c
static inline update_state_t timing_apply_pre_spike(
    uint32_t time, pre_trace_t trace, uint32_t last_pre_time,
    pre_trace_t last_pre_trace, uint32_t last_post_time,
    post_trace_t last_post_trace, update_state_t previous_state) {

    // Get time of event relative to last post-synaptic event
    uint32_t time_since_last_post = time - last_post_time;
    if (time_since_last_post > 0) {
        int32_t decayed_o1 = STDP_FIXED_MUL_16X16(
            last_post_trace, DECAY_LOOKUP_TAU_MINUS(time_since_last_post));

        log_debug("\t\ttime_since_last_post_event=%u, decayed_o1=%d\n",
            time_since_last_post, decayed_o1);

        // Apply depression to state (which is a weight state)
        return weight_one_term_apply_depression(previous_state, decayed_o1);
    } else {
        return previous_state;
    }
}
```
static inline update_state_t timing_apply_post_spike(
    uint32_t time, post_trace_t trace, uint32_t last_pre_time,
    pre_trace_t last_pre_trace, uint32_t last_post_time,
    post_trace_t last_post_trace, update_state_t previous_state) {

    // Get time of event relative to last pre-synaptic event
    uint32_t time_since_last_pre = time - last_pre_time;
    if (time_since_last_pre > 0) {
        int32_t decayed_r1 = STDP_FIXED_MUL_16X16(
            last_pre_trace, DECAY_LOOKUP_TAU_PLUS(time_since_last_pre));

        log_debug("\tt\tttime_since_last_pre_event=%u, decayed_r1=%d\n",
            time_since_last_pre, decayed_r1);

        // Apply potentiation to state (which is a weight_state)
        return weight_one_term_apply_potentiation(previous_state, decayed_r1);
    } else {
        return previous_state;
    }
}
Weight Dependence Rule

Translate relative spike timing into weight update

Python:
- Provide user interface via PyNN
- Initialise and write rule parameters for loading to machine
- Extend AbstractWeightDependence

C:
- Compile into application binary
- Implement potentiation/depression and data-handling
- Follow weight_dependence.c/h
class WeightDependenceAdditive(AbstractWeightDependence, AbstractHasAPlusAMinus):
    def __init__(self, w_min=0.0, w_max=1.0):
        AbstractWeightDependence.__init__(self)
        AbstractHasAPlusAMinus.__init__(self)
        self._w_min = w_min
        self._w_max = w_max

@property
def w_min(self):
    return self._w_min

@property
def w_max(self):
    return self._w_max

@property
def vertex_executable_suffix(self):
    return "additive"
weight_dependence.py

Specify memory requirements

- Four parameters: $W_{\text{min}}, W_{\text{max}}, A^+, A^-$
- 32-bit parameters (4 bytes)

```python
def get_parameters_sdram_usage_in_bytes(
    self, n_synapse_types, n_weight_terms):
    if n_weight_terms == 1:
        return (4 * 4) * n_synapse_types
    else:
        raise NotImplementedError(
            "Additive weight dependence only supports one term")
```
def write_parameters(
    self, spec, machine_time_step, weight_scales, n_weight_terms):

    # Loop through each synapse type's weight scale
    for w in weight_scales:

        # Scale the weights
        spec.write_value(
            data=int(round(self._w_min * w)), data_type=DataType.INT32)
        spec.write_value(
            data=int(round(self._w_max * w)), data_type=DataType.INT32)

        # Based on http://data.andrewdavison.info/docs/PyNN/_modules/pyNN
        # /standardmodels/synapses.html
        # Pre-multiply A+ and A- by Wmax
        spec.write_value(
            data=int(round(self._a_plus * self._w_max * w)),
            data_type=DataType.INT32)
        spec.write_value(
            data=int(round(self._a_minus * self._w_max * w)),
            data_type=DataType.INT32)
Initialise rule parameters

```c
address_t weight_initialise(address_t address,
                           uint32_t *ring_buffer_to_input_buffer_left_shifts) {
    use(ring_buffer_to_input_buffer_left_shifts);

    log_info("weight_initialise: starting");
    log_info("STDP additive one-term weight dependance");

    // Copy plasticity region data from address
    int32_t *plasticity_word = (int32_t*) address;
    for (uint32_t s = 0; s < SYNAPSE_TYPE_COUNT; s++) {
        plasticity_weight_region_data[s].min_weight = *plasticity_word++;
        plasticity_weight_region_data[s].max_weight = *plasticity_word++;
        plasticity_weight_region_data[s].a2_plus = *plasticity_word++;
        plasticity_weight_region_data[s].a2_minus = *plasticity_word++;

        log_info("\tSynapse type %u: Min weight:%d, Max weight:%d, A2+:%d, A2-:%d",
                 s, plasticity_weight_region_data[s].min_weight,
                 plasticity_weight_region_data[s].max_weight,
                 plasticity_weight_region_data[s].a2_plus,
                 plasticity_weight_region_data[s].a2_minus);
    }
    log_info("weight_initialise: completed successfully");

    // Return end address of region
    return (address_t) plasticity_word;
}
```
Get initial weight

```c
static inline weight_state_t weight_get_initial(weight_t weight,
                                                index_t synapse_type) {

    return (weight_state_t ) {
        .initial_weight = (int32_t) weight,
        .a2_plus = 0,
        .a2_minus = 0,
        .weight_region = &plasticity_weight_region_data[synapse_type]
    };
}
```
Apply depression

```c
static inline weight_state_t weight_one_term_apply_depression(
    weight_state_t state, int32_t a2_minus) {
    state.a2_minus += a2_minus;
    return state;
}
```

Apply potentiation

```c
static inline weight_state_t weight_one_term_apply_potentiation(
    weight_state_t state, int32_t a2_plus) {
    state.a2_plus += a2_plus;
    return state;
}
```
Calculate final weight

```c
static inline weight_t weight_get_final(weight_state_t new_state) {

    // Scale potentiation and depression
    // **NOTE** A2+ and A2- are pre-scaled into weight format
    int32_t scaled_a2_plus = STDP_FIXED_MUL_16X16(
        new_state.a2_plus, new_state.weight_region->a2_plus);
    int32_t scaled_a2_minus = STDP_FIXED_MUL_16X16(
        new_state.a2_minus, new_state.weight_region->a2_minus);

    // Apply all terms to initial weight
    int32_t new_weight = new_state.initial_weight + scaled_a2_plus
                          - scaled_a2_minus;

    // Clamp new weight
    new_weight = MIN(new_state.weight_region->max_weight,
                     MAX(new_weight, new_state.weight_region->min_weight));

    log_debug("\told_weight:%u, a2:+%d, a2:-%d, scaled a2:+%d, scaled a2:-%d,"
              " new_weight:%d",
              new_state.initial_weight, new_state.a2_plus, new_state.a2_minus,
              scaled_a2_plus, scaled_a2_minus, new_weight);

    return (weight_t) new_weight;
}
```
Building binary

Create build directory

- Name according to neuron model and plasticity model
  - “IF_curr_exp_stdp_mad” +
  - “_” + “pair” +
  - “_” + “additive”
Building binary

Build C code via Makefile:

```
NEURON_MODEL = $(SOURCE_DIR)/neuron/models/neuron_model_lif_impl.c
NEURON_MODEL_H = $(SOURCE_DIR)/neuron/models/neuron_model_lif_impl.h
INPUT_TYPE_H = $(SOURCE_DIR)/neuron/input_types/input_type_current.h
THRESHOLD_TYPE_H = $(SOURCE_DIR)/neuron/threshold_types/threshold_type_static.h
SYNAPSE_TYPE_H = $(SOURCE_DIR)/neuron/synapse_types/synapse_types_exponential_impl.h

SYNAPSE_DYNAMICS = $(SOURCE_DIR)/neuron/plasticity/stdp/synapse_dynamics_stdp_mad_impl.c
TIMINGDEPENDENCE = $(SOURCE_DIR)/neuron/plasticity/stdp/timing_dependence/timing_nearest_pair_impl.c
TIMING_DEPENDENCE_H = $(SOURCE_DIR)/neuron/plasticity/stdp/timing_dependence/timing_nearest_pair_impl.h

WEIGHT_DEPENDENCE = $(SOURCE_DIR)/neuron/plasticity/stdp/weight_dependence/weight_additive_one_term_impl.c
WEIGHT_DEPENDENCE_H = $(SOURCE_DIR)/neuron/plasticity/stdp/weight_dependence/weight_additive_one_term_impl.h
```
MODELS = IF_curr_exp \
    IF_cond_exp \
    IZK_curr_exp \
    IZK_cond_exp \
    IF_curr_exp_dual \
    IF_curr_exp_stdp_mad_pair_additive \
    IF_curr_exp_stdp_mad_pair_multiplicative \
    IF_cond_exp_stdp_mad_pair_additive \
    IF_curr_exp_stdp_mad_recurrent_dual_fsm_multiplicative \
    IF_curr_exp_stdp_mad_vogels_2011_additive \
    IF_curr_delta \
    IF_curr_exp_ca2_adaptive \
    IF_curr_exp_stdp_nearest_pair_additive \
    IF_curr_exp_stdp_nearest_pair_multiplicative \
    IF_curr_exp_stdp_pfister_triplet_additive \
    IF_curr_exp_stdp_mad_pfister_triplet_additive \
    IF_cond_exp_stdp_mad_nearest_pair_additive

BUILD_DIRS := $(addprefix builds/, $(MODELS))

all: $(BUILD_DIRS)
    for d in $(BUILD_DIRS); do (cd $$d; "$(MAKE)") || exit $?; done

clean: $(BUILD_DIRS)
    for d in $(BUILD_DIRS); do (cd $$d; "$(MAKE)" clean) || exit $?; done
## sPyNNaker STDP Implementations

<table>
<thead>
<tr>
<th>Build</th>
<th>Neuron Model</th>
<th>Timing Rule</th>
<th>Weight Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF_cond_exp_stdp_mad_nearest_pair_additive</td>
<td>IF_cond_exp</td>
<td>timing_nearest_pair_impl</td>
<td>weight_additive_one_term_impl</td>
</tr>
<tr>
<td>IF_cond_exp_stdp_mad_pair_additive</td>
<td>IF_cond_exp</td>
<td>timing_pair_impl</td>
<td>weight_additive_one_term_impl</td>
</tr>
<tr>
<td>IF_curr_exp_stdp_mad_nearest_pair_additive</td>
<td>IF_curr_exp</td>
<td>timing_nearest_pair_impl</td>
<td>weight_additive_one_term_impl</td>
</tr>
<tr>
<td>IF_curr_exp_stdp_mad_nearest_pair_multiplicative</td>
<td>IF_curr_exp</td>
<td>timing_nearest_pair_impl</td>
<td>weight_multiplicative_impl</td>
</tr>
<tr>
<td>IF_curr_exp_stdp_mad_pair_additive</td>
<td>IF_curr_exp</td>
<td>timing_pair_impl</td>
<td>weight_additive_one_term_impl</td>
</tr>
<tr>
<td>IF_curr_exp_stdp_mad_pair_multiplicative</td>
<td>IF_curr_exp</td>
<td>timing_pair_impl</td>
<td>weight_multiplicative_impl</td>
</tr>
<tr>
<td>IF_curr_exp_stdp_mad_pfister_triplet_additive</td>
<td>IF_curr_exp</td>
<td>timing_pfister_triplet_impl</td>
<td>weight_additive_two_term_impl</td>
</tr>
<tr>
<td>IF_curr_exp_stdp_mad_recurrent_dual_fsm_multiplicative</td>
<td>IF_curr_exp</td>
<td>timing_recurrent_dual_fsm_impl</td>
<td>weight_multiplicative_impl</td>
</tr>
<tr>
<td>IF_curr_exp_stdp_mad_vogels_2011_additive</td>
<td>IF_curr_exp</td>
<td>timing_vogels_2011_impl</td>
<td>weight_additive_one_term_impl</td>
</tr>
<tr>
<td>IF_curr_exp_stdp_recurrent_pre_stochastic_multiplicative</td>
<td>IF_curr_exp</td>
<td>timing_recurrent_pre_stochastic_impl</td>
<td>weight_multiplicative_impl</td>
</tr>
</tbody>
</table>
Synapse Dynamics

Two single-neuron populations connected by a single STDP synapse:

- Potentiation/depression performed by post-synaptic core
- Framework defined via `synapse_dynamics_stdp_mad_impl`

Pre-event sensitive system
Synapse Structure

Python class to read/write weight structure

- ‘Weight only’ suitable for most applications
- Possible to add additional data structures
- Constructed by timing dependence __init__

```python
class SynapseStructureWeightOnly(AbstractSynapseStructure):

    def __init__(self):
        AbstractSynapseStructure.__init__(self)

    def get_n_bytes_per_connection(self):
        return 2

    def get_synaptic_data(self, connections):
        plastic_plastic = numpy.rint(
            numpy.abs(connections["weight"])).astype("uint16")
        return plastic_plastic.view(dtype="uint8").reshape((-1, 2))

    def read_synaptic_data(self, fp_size, pp_data):
        return numpy.concatenate([pp_data[i].view(dtype="uint16")[:fp_size[i]]
                                  for i in range(len(pp_data))])
```
// Plastic synapse types are just weights;
typedef weight_t plastic_synapse_t;

// The update state is purely a weight state
typedef weight_state_t update_state_t;

// The final state is just a weight as this is both the weight and the synaptic word
typedef weight_t final_state_t;

//---------------------------------------
static inline update_state_t synapse_structure_get_update_state(
    plastic_synapse_t synaptic_word, index_t synapse_type) {
    return weight_get_initial(synaptic_word, synapse_type);
}

static inline final_state_t synapse_structure_get_final_state(update_state_t state) {
    return weight_get_final(state);
}

static inline weight_t synapse_structure_get_final_weight(final_state_t final_state) {
    return final_state;
}

static inline plastic_synapse_t synapse_structure_get_final_synaptic_word(final_state_t final_state) {
    return final_state;
}
Overall Framework

- timing_dependence_spike_pair.py
- weight_dependence_additive.py
- synapse_dynamics_stdp.py

- synapse_structure_weight_only.py

- timing_pair_impl.h/c
- weight_dependence_one_term_additive_impl.h/c
- synapse_dynamics_stdp_mad_impl.c

- synapse_structure_weight_only_impl.h

- Called via PyNN
- Called by sPyNNaker toolchain

- Specified in Makefile
Summary

- sPyNNaker - flexible STDP framework
- Separate timing and weight update rules available
- Python/C implementation
  - Python: PyNN interface and create datastructures
  - C: load data and define runtime update rules
- Variety of existing examples demonstrate implementation
  - Good starting point for a new implementation
Coming Soon: Neuromodulation

- Classic Hebbian learning puts emphasis on pre- and post-synaptic activity
  - Neglects effect of neuromodulators, such as dopamine responsible for **reward learning**
- Three-factor neo-Hebbian learning rules
  - Neuromodulation signal M, conveying reward or novelty, which affects STDP.

\[
\begin{align*}
\text{Hebbian} & : \quad \dot{w} = H(pre, post) \\
\text{neo-Hebbian three-factor} & : \quad \dot{w} = H'(pre, post, M)
\end{align*}
\]
Coming Soon: Synaptic Rewiring

Rising amount of interest for structural plasticity.

Collaboration between PyNN, Nest, BrainScaleS and SpiNNaker to standardise high-level description.

Real-time simulation of synaptic rewiring:
- **Topographic map formation**
- **Homeostasis**

S. Bamford et al. “Synaptic rewiring for topographic mapping and receptive field development”
M. Butz, A. van Ooyen “A Simple Rule for Dendritic Spine and Axonal Bouton Formation Can Account for Cortical Reorganization after Focal Retinal Lesions”
Amendment to Lab Manual

f. Now add similar import statements to the sPyNNaker8 repository. Find `spynnaker8/__init__.py` and add:

```python
from spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.
    weight_dependence_additive_custom
import WeightDependenceAdditiveCustom as
AdditiveWeightDependenceCustom

__all__ = [...,
    # plastic stuff
    'STDPMechanism', 'AdditiveWeightDependence',
    'AdditiveWeightDependenceCustom', 'MultiplicativeWeightDependence',
    'SpikePairRule', 'SpikePairRuleCustom', ...]
```

2. Now update the C Code
   a. Find `weight_additive_one_term_impl.h` and `weight_additive_one_term_impl.c` and copy to new names in the same directory: e.g. `weight_additive_one_term_custom_impl.h` and