sPyNNaker Documentation

Jan 29, 2020

Contents

These pages document the python code for the sPyNNaker module which is part of the SpiNNaker Project.

This code depends on SpiNNUtils, SpiNNMachine, SpiNNStorageHandlers, SpiNNMan, PACMAN, DataSpecification, SpiNNFrontEndCommon (Combined_documentation).

Contents:

CHAPTER 1

spynnaker package

1.1 Subpackages

1.1.1 spynnaker.pyNN package

1.1.1.1 Subpackages

spynnaker.pyNN.connections package

Submodules

spynnaker.pyNN.connections.ethernet_command_connection module

spynnaker.pyNN.connections.ethernet_control_connection module

class spynnaker.pyNN.connections.ethernet_control_connection.**EthernetControlConnection**(*trans lo- cal_h lo- cal_p*

Bases: spinnman.connections.udp_packet_connections.eieio_connection.EIEIOConnection

A connection that can translate Ethernet control messages received from a Population

Parameters

- **translator** – The translator of multicast to control commands
- **local_host** – The optional host to listen on
- **local_port** – The optional port to listen on

close()
Close the connection

run()

spynnaker.pyNN.connections.spynnaker_live_spikes_connection module

class spynnaker.pyNN.connections.spynnaker_live_spikes_connection.**SpynnakerLiveSpikesConne**

Bases: spinn_front_end_common.utilities.connections.live_event_connection.LiveEventConnection

A connection for receiving and sending live spikes from and to SpiNNaker

Parameters

- **receive_labels** (*iterable of str*) – Labels of population from which live spikes will be received.
- **send_labels** (*iterable of str*) – Labels of population to which live spikes will be sent
- **local_host** (*str*) – Optional specification of the local hostname or IP address of the interface to listen on
- **local_port** (*int*) – Optional specification of the local port to listen on. Must match the port that the toolchain will send the notification on (19999 by default)

send_spike(*label*, *neuron_id*, *send_full_keys=False*)
Send a spike from a single neuron

Parameters

- **label** (*str*) – The label of the population from which the spike will originate
- **neuron_id** (*int*) – The ID of the neuron sending a spike
- **send_full_keys** (*bool*) – Determines whether to send full 32-bit keys, getting the key for each neuron from the database, or whether to send 16-bit neuron IDs directly

send_spikes(*label*, *neuron_ids*, *send_full_keys=False*)
Send a number of spikes

Parameters

- **label** (*str*) – The label of the population from which the spikes will originate
- **neuron_ids** (*list(int)*) – array-like of neuron IDs sending spikes
- **send_full_keys** (*bool*) – Determines whether to send full 32-bit keys, getting the key for each neuron from the database, or whether to send 16-bit neuron IDs directly

spynnaker.pyNN.connections.spynnaker_poisson_control_connection module

Module contents

spynnaker.pyNN.external_devices_models package

Subpackages

spynnaker.pyNN.external_devices_models.push_bot package

Subpackages

spynnaker.pyNN.external_devices_models.push_bot.push_bot_control_modules package

Submodules

spynnaker.pyNN.external_devices_models.push_bot.push_bot_control_modules.push_bot_lif_ethernet module

spynnaker.pyNN.external_devices_models.push_bot.push_bot_control_modules.push_bot_lif_spinnaker_link module

Module contents

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet package

Submodules

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_device module

class `spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_d`

Bases: *spynnaker.pyNN.external_devices_models.abstract_multicast_controllable_device.AbstractMulticastControllableDevice*

An arbitrary PushBot device

Parameters

- **protocol** – The protocol instance to get commands from
- **device** – The Enum instance of the device to control
- **uses_payload** – True if the device uses a payload for control

device_control_key

The key that must be sent to the device to control it

Return type int

device_control_max_value

The maximum value to send to the device

Return type float

device_control_min_value
The minimum value to send to the device

Return type float

device_control_partition_id
A partition ID to give to an outgoing edge partition that will control this device

Return type str

device_control_send_type
The type of data to be sent.

Return type *SendType*

device_control_timesteps_between_sending
The number of timesteps between sending commands to the device. This defines the "sampling interval" for the device.

Return type int

device_control_uses_payload
True if the control of the device accepts an arbitrary valued payload, the value of which will change the devices behaviour

Return type bool

protocol
The protocol instance, for use in the subclass

set_command_protocol(*command_protocol*)
Set the protocol use to send setup and shutdown commands, separately from the protocol used to control the device

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_laser_device module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_l

Bases: *spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_device.PushBotEthernetDevice*, spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

The Laser of a PushBot

Parameters

- **laser** – The PushBotLaser value to control
- **protocol** – The protocol instance to get commands from
- **start_active_time** – The "active time" value to send at the start

- **start_total_period** – The "total period" value to send at the start
- **start_frequency** – The "frequency" to send at the start
- **timesteps_between_send** – The number of timesteps between sending commands to the device, or None to use the default

pause_stop_commands
The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

set_command_protocol (*command_protocol*)
Set the protocol use to send setup and shutdown commands, separately from the protocol used to control the device

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands
The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_led_device module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_l

Bases: *spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_device.PushBotEthernetDevice*, spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

The LED of a PushBot

Parameters

- **led** – The PushBotLED parameter to control
- **protocol** – The protocol instance to get commands from
- **start_active_time_front** – The "active time" to set for the front LED at the start
- **start_active_time_back** – The "active time" to set for the back LED at the start
- **start_total_period** – The "total period" to set at the start
- **start_frequency** – The "frequency" to set at the start

- **timesteps_between_send** – The number of timesteps between sending commands to the device, or None to use the default

pause_stop_commands
The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

set_command_protocol (*command_protocol*)
Set the protocol use to send setup and shutdown commands, separately from the protocol used to control the device

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands
The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_motor_device module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_r

Bases: *spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_device.PushBotEthernetDevice*, spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

The motor of a PushBot

Parameters

- **motor** – a PushBotMotor value to indicate the motor to control
- **protocol** – The protocol used to control the device
- **timesteps_between_send** – The number of timesteps between sending commands to the device, or None to use the default

pause_stop_commands
The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

set_command_protocol (*command_protocol*)
Set the protocol use to send setup and shutdown commands, separately from the protocol used to control the device

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands
The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_retina_device module

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_ethernet_speaker_device module

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_retina_connection module

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_translator module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_translato

Bases: *spynnaker.pyNN.external_devices_models.abstract_ethernet_translator.AbstractEthernetTranslator*

Translates packets between PushBot Multicast packets and PushBot Wi-Fi Commands

Parameters

- **protocol** – The instance of the PushBot protocol to get keys from
- **pushbot_wifi_connection** – A Wi-Fi connection to the PushBot

translate_control_packet (*multicast_packet*)
Translate a multicast packet received over Ethernet and send appropriate messages to the external device

Parameters **multicast_packet** (spinnman.messages.eieio.data_messages.AbstractEIEIODataElement) – A received multicast packet

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_wifi_connection module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_wifi_conne

Bases: spinnman.connections.abstract_classes.connection.Connection, spinnman.connections.abstract_classes.listenable.Listenable

A connection to a PushBot via Wi-Fi.

Parameters

- **remote_host** (*str*) – The IP address of the PushBot
- **remote_port** (*int*) – The port number of the PushBot (default 56000)

Raises **spinnman.exceptions.SpinnmanIOException** – If there is an error setting up the communication channel

close()
See `spinnman.connections.Connection.close()`

get_receive_method()
Get the method that receives for this connection

is_connected()
See `spinnman.connections.Connection.is_connected()`

is_ready_to_receive(*timeout=0*)
Determines if there is an SCP packet to be read without blocking

Parameters **timeout** (`int`) – The time to wait before returning if the connection is not ready

Returns True if there is an SCP packet to be read

Return type bool

local_ip_address
The local IP address to which the connection is bound.

Returns The local IP address as a dotted string, e.g. *0.0.0.0*

Return type str

Raises **None** – No known exceptions are thrown

local_port
The local port to which the connection is bound.

Returns The local port number

Return type int

Raises **None** – No known exceptions are thrown

receive(*timeout=None*)
Receive data from the connection

Parameters **timeout** (`float or None`) – The timeout, or None to wait forever

Returns The data received

Return type bytestring

Raises

- **SpinnmanTimeoutException** – If a timeout occurs before any data is received
- **SpinnmanIOException** – If an error occurs receiving the data

remote_ip_address
The remote IP address to which the connection is connected.

Returns The remote IP address as a dotted string, or None if not connected remotely

Return type str

remote_port
The remote port to which the connection is connected.

Returns The remote port, or None if not connected remotely

Return type int

send(*data*)
Send data down this connection

Parameters **data** (*bytestring*) – The data to be sent

Raises **SpinnmanIOException** – If there is an error sending the data

spynnaker.pyNN.external_devices_models.push_bot.push_bot_ethernet.push_bot_wifi_connection

Get an existing connection to a PushBot, or make a new one.

Parameters

- **remote_host** (*str*) – The IP address of the PushBot
- **remote_port** (*int*) – The port number of the PushBot (default 56000)

Module contents

spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters package

Submodules

spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_laser module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_laser.**P**

Bases: *spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice*

An enumeration.

LASER_ACTIVE_TIME = 1

LASER_FREQUENCY = 2

LASER_TOTAL_PERIOD = 0

spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_led module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_led.**Pusl**

Bases: *spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice*

An enumeration.

LED_BACK_ACTIVE_TIME = 2

LED_FREQUENCY = 3

LED_FRONT_ACTIVE_TIME = 1

LED_TOTAL_PERIOD = 0

spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_motor module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_motor.**Pu**

Bases: *spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice*

An enumeration.

MOTOR_0_LEAKY = 1

MOTOR_0_PERMANENT = 0

MOTOR_1_LEAKY = 3

MOTOR_1_PERMANENT = 2

spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_retina_resolution module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_retina_r

Bases: enum.Enum

An enumeration.

DOWNSAMPLE_16_X_16 = <RetinaKey.DOWNSAMPLE_16_X_16: 268435456>

DOWNSAMPLE_32_X_32 = <RetinaKey.DOWNSAMPLE_32_X_32: 201326592>

DOWNSAMPLE_64_X_64 = <RetinaKey.DOWNSAMPLE_64_X_64: 134217728>

NATIVE_128_X_128 = <RetinaKey.NATIVE_128_X_128: 67108864>

spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_retina_viewer module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_retina_v

Bases: threading.Thread

local_host

local_port

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_speaker module

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.push_bot_speaker

Bases: *spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice*

An enumeration.

SPEAKER_ACTIVE_TIME = 1

SPEAKER_MELODY = 3

SPEAKER_TONE = 2

SPEAKER_TOTAL_PERIOD = 0

Module contents

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.**PushBotLaser**

Bases: *spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice*

An enumeration.

LASER_ACTIVE_TIME = 1

LASER_FREQUENCY = 2

LASER_TOTAL_PERIOD = 0

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.**PushBotLED**

Bases: *spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice*

An enumeration.

LED_BACK_ACTIVE_TIME = 2

LED_FREQUENCY = 3

LED_FRONT_ACTIVE_TIME = 1

LED_TOTAL_PERIOD = 0

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.**PushBotMotor**

Bases: *spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice*

An enumeration.

MOTOR_0_LEAKY = 1

MOTOR_0_PERMANENT = 0

MOTOR_1_LEAKY = 3

MOTOR_1_PERMANENT = 2

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.**PushBotSpeaker**

Bases: *spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice*

An enumeration.

SPEAKER_ACTIVE_TIME = 1

SPEAKER_MELODY = 3

SPEAKER_TONE = 2

SPEAKER_TOTAL_PERIOD = 0

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.**PushBotRetinaReso**

Bases: enum.Enum

An enumeration.

DOWNSAMPLE_16_X_16 = <RetinaKey.DOWNSAMPLE_16_X_16: 268435456>

DOWNSAMPLE_32_X_32 = <RetinaKey.DOWNSAMPLE_32_X_32: 201326592>

DOWNSAMPLE_64_X_64 = <RetinaKey.DOWNSAMPLE_64_X_64: 134217728>

NATIVE_128_X_128 = <RetinaKey.NATIVE_128_X_128: 67108864>

class spynnaker.pyNN.external_devices_models.push_bot.push_bot_parameters.**PushBotRetinaView**

Bases: threading.Thread

local_host

local_port

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

spynnaker.pyNN.external_devices_models.push_bot.push_bot_spinnaker_link package

Submodules

spynnaker.pyNN.external_devices_models.push_bot.push_bot_spinnaker_link.push_bot_spinnaker_link_laser_de module

spynnaker.pyNN.external_devices_models.push_bot.push_bot_spinnaker_link.push_bot_spinnaker_link_led_devi module

spynnaker.pyNN.external_devices_models.push_bot.push_bot_spinnaker_link.push_bot_spinnaker_link_motor_d module

spynnaker.pyNN.external_devices_models.push_bot.push_bot_spinnaker_link.push_bot_spinnaker_link_retina_de module

spynnaker.pyNN.external_devices_models.push_bot.push_bot_spinnaker_link.push_bot_spinnaker_link_speaker_ module

Module contents

Submodules

spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device module

class spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.**Abst**

Bases: enum.Enum

An enumeration.

max_value

min_value

protocol_property

send_type

time_between_send

spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_retina_device module

class spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_retina_device.**Abst**

Bases: spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

pause_stop_commands

The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

start_resume_commands

The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands

The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

Module contents

class spynnaker.pyNN.external_devices_models.push_bot.**AbstractPushBotOutputDevice**

Bases: enum.Enum

An enumeration.

max_value

min_value

protocol_property

send_type

time_between_send

class spynnaker.pyNN.external_devices_models.push_bot.**AbstractPushBotRetinaDevice**(*protocol*, *resolution*)

Bases: spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

pause_stop_commands
The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands
The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

Submodules

spynnaker.pyNN.external_devices_models.abstract_ethernet_controller module

class spynnaker.pyNN.external_devices_models.abstract_ethernet_controller.**AbstractEthernetC**

Bases: object

A controller that can send multicast packets which can be received over Ethernet and translated to control an external device

get_external_devices()
Get the external devices that are to be controlled by the controller

get_message_translator()
Get the translator of messages

Return type *spynnaker.pyNN.external_devices_models.AbstractEthernetTranslator*

get_outgoing_partition_ids()
Get the partition IDs of messages coming out of the controller

Return type list(str)

spynnaker.pyNN.external_devices_models.abstract_ethernet_sensor module

class spynnaker.pyNN.external_devices_models.abstract_ethernet_sensor.**AbstractEthernetSenso**

Bases: `object`

get_database_connection()
: Get a Database Connection instance that this device uses to inject packets

get_injector_label()
: Get the label to give to the Spike Injector

get_injector_parameters()
: Get the parameters of the Spike Injector to use with this device

get_n_neurons()
: Get the number of neurons that will be sent out by the device

get_translator()
: Get a translator of multicast commands to Ethernet commands

spynnaker.pyNN.external_devices_models.abstract_ethernet_translator module

class spynnaker.pyNN.external_devices_models.abstract_ethernet_translator.**AbstractEthernetT**

Bases: `object`

A module that can translate packets received over Ethernet into control of an external device

translate_control_packet(*multicast_packet*)
: Translate a multicast packet received over Ethernet and send appropriate messages to the external device

Parameters **multicast_packet** (`spinnman.messages.eieio.data_messages.AbstractEIEIODataElement`) – A received multicast packet

spynnaker.pyNN.external_devices_models.abstract_multicast_controllable_device module

class spynnaker.pyNN.external_devices_models.abstract_multicast_controllable_device.**Abstrac**

Bases: `object`

A device that can be controlled by sending multicast packets to it, either directly, or via Ethernet using an AbstractEthernetTranslator

device_control_key
: The key that must be sent to the device to control it

Return type int

device_control_max_value
: The maximum value to send to the device

Return type float

device_control_min_value
: The minimum value to send to the device

Return type float

device_control_partition_id
: A partition ID to give to an outgoing edge partition that will control this device

Return type str

device_control_scaling_factor
The scaling factor used to send the payload to this device.

Return type int

device_control_send_type
The type of data to be sent.

Return type *SendType*

device_control_timesteps_between_sending
The number of timesteps between sending commands to the device. This defines the "sampling interval" for the device.

Return type int

device_control_uses_payload
True if the control of the device accepts an arbitrary valued payload, the value of which will change the devices behaviour

Return type bool

class spynnaker.pyNN.external_devices_models.abstract_multicast_controllable_device.**SendTyp**
Bases: enum.Enum

The data type to be sent in the payload of the multicast packet

SEND_TYPE_ACCUM = 2

SEND_TYPE_FRACT = 4

SEND_TYPE_INT = 0

SEND_TYPE_UACCUM = 3

SEND_TYPE_UFRACT = 5

SEND_TYPE_UINT = 1

spynnaker.pyNN.external_devices_models.arbitrary_fpga_device module

class spynnaker.pyNN.external_devices_models.arbitrary_fpga_device.**ArbitraryFPGADevice**(*n_ne fpga_ fpga_ boar la- bel=*
Bases: pacman.model.graphs.application.application_fpga_vertex.ApplicationFPGAVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

spynnaker.pyNN.external_devices_models.external_device_lif_control module

class spynnaker.pyNN.external_devices_models.external_device_lif_control.**ExternalDeviceLif**
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Abstract control module for the PushBot, based on the LIF neuron, but without spikes, and using the voltage as the output to the various devices

create_vertex(*n_neurons*, *label*, *constraints*, *spikes_per_second*, *ring_buffer_sigma*, *incoming_spike_buffer_size*)
Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

spynnaker.pyNN.external_devices_models.external_device_lif_control_vertex module

class spynnaker.pyNN.external_devices_models.external_device_lif_control_vertex.**ExternalDev**

Bases: *spynnaker.pyNN.models.neuron.abstract_population_vertex.AbstractPopulationVertex*, *spynnaker.pyNN.external_devices_models.abstract_ethernet_controller.AbstractEthernetController*, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraint AbstractProvidesOutgoingPartitionConstraints, spinn_front_end_common.abstract_models.abstract_vertex_with_dependent_vertices.AbstractVertexWithEdgeToDependentVertices

Abstract control module for the pushbot, based on the LIF neuron, but without spikes, and using the voltage as the output to the various devices

Parameters

- **n_neurons** – The number of neurons in the population
- **devices** – The AbstractMulticastControllableDevice instances to be controlled by the population
- **create_edges** – True if edges to the devices should be added by this dev (set to False if using the dev over Ethernet using a translator)

- **translator** – Translator to be used when used for Ethernet communication. Must be provided if the dev is to be controlled over Ethernet.

dependent_vertices()
Return the vertices which this vertex depends upon

Return type iterable(ApplicationVertex)

edge_partition_identifiers_for_dependent_vertex(*vertex*)
Return the dependent edge identifiers for a particular dependent vertex.

Parameters **vertex** (*ApplicationVertex*) –

Return type iterable(str)

get_external_devices()
Get the external devices that are to be controlled by the controller

get_message_translator()
Get the translator of messages

Return type *spynnaker.pyNN.external_devices_models.AbstractEthernetTranslator*

get_outgoing_partition_constraints(*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters **partition** (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

get_outgoing_partition_ids()
Get the partition IDs of messages coming out of the controller

Return type list(str)

routing_key_partition_atom_mapping(*routing_info*, *partition*)
Returns a list of atom to key mapping.

Parameters

- **routing_info** (*RoutingInfo*) – the routing info object to consider
- **partition** (*AbstractOutgoingEdgePartition*) – the routing partition to handle.

Returns a iterable of tuples of atom IDs to keys.

Return type iterable(tuple(int,int))

spynnaker.pyNN.external_devices_models.external_spinnaker_link_cochlea_device module

class spynnaker.pyNN.external_devices_models.external_spinnaker_link_cochlea_device.**Externa**

Bases: pacman.model.graphs.application.application_spinnaker_link_vertex.

ApplicationSpiNNakerLinkVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

spynnaker.pyNN.external_devices_models.external_spinnaker_link_fpga_retina_device module

class spynnaker.pyNN.external_devices_models.external_spinnaker_link_fpga_retina_device.**Ext**

Bases: pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex, spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.AbstractProvidesOutgoingPartitionConstraints, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

Parameters

- **mode** – The retina "mode"
- **retina_key** – The value of the top 16-bits of the key
- **spinnaker_link_id** – The SpiNNaker link to which the retina is connected
- **polarity** – The "polarity" of the retina data
- **label** –
- **board_address** –

DOWN_POLARITY = 'DOWN'

MERGED_POLARITY = 'MERGED'

MODE_128 = '128'

MODE_16 = '16'

MODE_32 = '32'

MODE_64 = '64'

UP_POLARITY = 'UP'

static get_n_neurons(*mode*, *polarity*)

get_outgoing_partition_constraints(*partition*)

Get constraints to be added to the given edge that comes out of this vertex.

Parameters **partition** (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

pause_stop_commands
The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands
The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

spynnaker.pyNN.external_devices_models.external_spinnaker_link_fpga_retina_device.**get_spike**

spynnaker.pyNN.external_devices_models.external_spinnaker_link_fpga_retina_device.**get_x_fro**

spynnaker.pyNN.external_devices_models.external_spinnaker_link_fpga_retina_device.**get_y_fro**

spynnaker.pyNN.external_devices_models.munich_spinnaker_link_motor_device module

class spynnaker.pyNN.external_devices_models.munich_spinnaker_link_motor_device.**MunichMoto**

Bases: pacman.model.graphs.application.application_vertex.ApplicationVertex, spinn_front_end_common.abstract_models.abstract_vertex_with_dependent_vertices.AbstractVertexWithEdgeToDependentVertices, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification, spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraint AbstractProvidesOutgoingPartitionConstraints, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

An Omnibot motor control device - has a real vertex and an external device vertex

PARAMS_REGION = 1

PARAMS_SIZE = 28

SYSTEM_REGION = 0

create_machine_vertex(*vertex_slice*, *resources_required*, *label=None*, *constraints=None*)
Create a machine vertex from this application vertex

Parameters

- **vertex_slice** (*Slice*) – The slice of atoms that the machine vertex will cover
- **resources_required** (*ResourceContainer*) – the resources used by the machine vertex
- **label** (*str or None*) – human readable label for the machine vertex
- **constraints** (*iterable(AbstractConstraint)*) – Constraints to be passed on to the machine vertex

default_initial_values = {}

default_parameters = {'board_address': None, 'continue_if_not_different': True, 'dela

dependent_vertices()
Return the vertices which this vertex depends upon

Return type iterable(ApplicationVertex) Return the vertices which this vertex depends upon

edge_partition_identifiers_for_dependent_vertex(*vertex*)
Return the dependent edge identifiers for a particular dependent vertex.

Parameters **vertex** (*ApplicationVertex*) –

Return type iterable(str) Return the dependent edge identifier

generate_data_specification(*spec*, *placement*, *routing_info*, *machine_time_step*, *time_scale_factor*)
Generate a data specification.

Parameters

- **spec** (*DataSpecificationGenerator*) – The data specification to write to
- **placement** (*Placement*) – the placement the vertex is located at

Return type None

get_binary_file_name()
Get the binary name to be run for this vertex.

Return type str

get_binary_start_type()
Get the start type of the binary to be run.

Return type ExecutableType

get_outgoing_partition_constraints(*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters **partition** (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

get_resources_used_by_atoms(*vertex_slice*)
Get the separate resource requirements for a range of atoms

Parameters **vertex_slice** (*Slice*) – the low value of atoms to calculate resources from

Returns a Resource container that contains a CPUCyclesPerTickResource, DTCMResource and SDRAMResource

Return type ResourceContainer

Raises **None** – this method does not raise any known exception

n_atoms
The number of atoms in the vertex

Return type int

reserve_memory_regions (*spec*)
Reserve SDRAM space for memory areas: 1) Area for information on what data to record 2) area for start commands 3) area for end commands

spynnaker.pyNN.external_devices_models.munich_spinnaker_link_retina_device module

class spynnaker.pyNN.external_devices_models.munich_spinnaker_link_retina_device.**MunichRet**

Bases: pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex, spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.AbstractProvidesOutgoingPartitionConstraints, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

DOWN_POLARITY = 'DOWN'

LEFT_RETINA = 'LEFT'

LEFT_RETINA_DISABLE = 69

LEFT_RETINA_ENABLE = 69

LEFT_RETINA_KEY_SET = 67

MANAGEMENT_BIT = 1024

MANAGEMENT_MASK = 4294965248

MERGED_POLARITY = 'MERGED'

RIGHT_RETINA = 'RIGHT'

RIGHT_RETINA_DISABLE = 70

RIGHT_RETINA_ENABLE = 70

RIGHT_RETINA_KEY_SET = 68

UP_POLARITY = 'UP'

default_parameters = {'board_address': None, 'label': 'MunichRetinaDevice', 'polarit

get_outgoing_partition_constraints (*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters partition (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

pause_stop_commands
The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands
The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

spynnaker.pyNN.external_devices_models.munich_spinnaker_link_retina_device.**get_spike_value_**

spynnaker.pyNN.external_devices_models.munich_spinnaker_link_retina_device.**get_x_from_robot**

spynnaker.pyNN.external_devices_models.munich_spinnaker_link_retina_device.**get_y_from_robot**

spynnaker.pyNN.external_devices_models.threshold_type_multicast_device_control module

class spynnaker.pyNN.external_devices_models.threshold_type_multicast_device_control.**Thresh**
Bases: *spynnaker.pyNN.models.neuron.threshold_types.abstract_threshold_type.AbstractThresholdType*

A threshold type that can send multicast keys with the value of membrane voltage as the payload

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters state_variables (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters n_neurons (*int*) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (*str*) – The name of the variable

Return type bool

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

Module contents

class spynnaker.pyNN.external_devices_models.**AbstractEthernetController**
Bases: object

A controller that can send multicast packets which can be received over Ethernet and translated to control an external device

get_external_devices ()
Get the external devices that are to be controlled by the controller

get_message_translator ()
Get the translator of messages

Return type *spynnaker.pyNN.external_devices_models.AbstractEthernetTranslator*

get_outgoing_partition_ids ()
Get the partition IDs of messages coming out of the controller

Return type list(str)

class spynnaker.pyNN.external_devices_models.**AbstractEthernetSensor**
Bases: object

get_database_connection ()
Get a Database Connection instance that this device uses to inject packets

get_injector_label()
Get the label to give to the Spike Injector

get_injector_parameters()
Get the parameters of the Spike Injector to use with this device

get_n_neurons()
Get the number of neurons that will be sent out by the device

get_translator()
Get a translator of multicast commands to Ethernet commands

class spynnaker.pyNN.external_devices_models.**AbstractEthernetTranslator**
Bases: object

A module that can translate packets received over Ethernet into control of an external device

translate_control_packet(*multicast_packet*)
Translate a multicast packet received over Ethernet and send appropriate messages to the external device

Parameters multicast_packet (spinnman.messages.eieio.data_messages.AbstractEIEIODataElement) – A received multicast packet

class spynnaker.pyNN.external_devices_models.**ArbitraryFPGADevice**(*n_neurons*, *fpga_link_id*, *fpga_id*, *board_address=None*, *label=None*)
Bases: pacman.model.graphs.application.application_fpga_vertex.ApplicationFPGAVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

class spynnaker.pyNN.external_devices_models.**AbstractMulticastControllableDevice**
Bases: object

A device that can be controlled by sending multicast packets to it, either directly, or via Ethernet using an AbstractEthernetTranslator

device_control_key
The key that must be sent to the device to control it

Return type int

device_control_max_value
The maximum value to send to the device

Return type float

device_control_min_value
The minimum value to send to the device

Return type float

device_control_partition_id
A partition ID to give to an outgoing edge partition that will control this device

Return type str

device_control_scaling_factor
The scaling factor used to send the payload to this device.

Return type int

device_control_send_type
The type of data to be sent.

Return type *SendType*

device_control_timesteps_between_sending
The number of timesteps between sending commands to the device. This defines the "sampling interval" for the device.

Return type int

device_control_uses_payload
True if the control of the device accepts an arbitrary valued payload, the value of which will change the devices behaviour

Return type bool

class spynnaker.pyNN.external_devices_models.**ExternalDeviceLifControl**(***kwargs*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Abstract control module for the PushBot, based on the LIF neuron, but without spikes, and using the voltage as the output to the various devices

create_vertex(*n_neurons, label, constraints, spikes_per_second, ring_buffer_sigma, incoming_spike_buffer_size*)
Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

class spynnaker.pyNN.external_devices_models.**ExternalCochleaDevice**(*n_neurons, spinnaker_link, label=None, board_address=None*)
Bases: pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

class spynnaker.pyNN.external_devices_models.**ExternalFPGARetinaDevice**(*mode, retina_key, spinnaker_link_id, polarity, label=None, board_address=None*)
Bases: pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex, spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.AbstractProvidesOutgoingPartitionConstraints, spinn_front_end_common.

abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

Parameters

- **mode** – The retina "mode"
- **retina_key** – The value of the top 16-bits of the key
- **spinnaker_link_id** – The SpiNNaker link to which the retina is connected
- **polarity** – The "polarity" of the retina data
- **label** –
- **board_address** –

DOWN_POLARITY = 'DOWN'

MERGED_POLARITY = 'MERGED'

MODE_128 = '128'

MODE_16 = '16'

MODE_32 = '32'

MODE_64 = '64'

UP_POLARITY = 'UP'

static get_n_neurons (*mode*, *polarity*)

get_outgoing_partition_constraints (*partition*)

Get constraints to be added to the given edge that comes out of this vertex.

Parameters partition (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

pause_stop_commands

The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

start_resume_commands

The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands

The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

class spynnaker.pyNN.external_devices_models.**MunichMotorDevice**(*spinnaker_link_id, board_address=None, speed=30, sample_time=4096, update_time=512, delay_time=5, delta_threshold=23, continue_if_not_different=True, label=None*)

Bases: pacman.model.graphs.application.application_vertex.ApplicationVertex, spinn_front_end_common.abstract_models.abstract_vertex_with_dependent_vertices.AbstractVertexWithEdgeToDependentVertices, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification, spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraint AbstractProvidesOutgoingPartitionConstraints, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

An Omnibot motor control device - has a real vertex and an external device vertex

PARAMS_REGION = 1

PARAMS_SIZE = 28

SYSTEM_REGION = 0

create_machine_vertex(*vertex_slice, resources_required, label=None, constraints=None*)

Create a machine vertex from this application vertex

Parameters

- **vertex_slice** (*Slice*) – The slice of atoms that the machine vertex will cover
- **resources_required** (*ResourceContainer*) – the resources used by the machine vertex
- **label** (*str or None*) – human readable label for the machine vertex
- **constraints** (*iterable(AbstractConstraint)*) – Constraints to be passed on to the machine vertex

default_initial_values = {}

default_parameters = {'board_address': None, 'continue_if_not_different': True, 'del

dependent_vertices()

Return the vertices which this vertex depends upon

Return type iterable(ApplicationVertex) Return the vertices which this vertex depends upon

edge_partition_identifiers_for_dependent_vertex(*vertex*)

Return the dependent edge identifiers for a particular dependent vertex.

Parameters **vertex** (*ApplicationVertex*) –

Return type iterable(str) Return the dependent edge identifier

generate_data_specification(*spec*, *placement*, *routing_info*, *machine_time_step*, *time_scale_factor*)
Generate a data specification.

Parameters

- **spec** (*DataSpecificationGenerator*) – The data specification to write to
- **placement** (*Placement*) – the placement the vertex is located at

Return type None

get_binary_file_name()
Get the binary name to be run for this vertex.

Return type str

get_binary_start_type()
Get the start type of the binary to be run.

Return type ExecutableType

get_outgoing_partition_constraints(*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters partition (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

get_resources_used_by_atoms(*vertex_slice*)
Get the separate resource requirements for a range of atoms

Parameters vertex_slice (*Slice*) – the low value of atoms to calculate resources from

Returns a Resource container that contains a CPUCyclesPerTickResource, DTCMResource and SDRAMResource

Return type ResourceContainer

Raises None – this method does not raise any known exception

n_atoms
The number of atoms in the vertex

Return type int

reserve_memory_regions(*spec*)
Reserve SDRAM space for memory areas: 1) Area for information on what data to record 2) area for start commands 3) area for end commands

class spynnaker.pyNN.external_devices_models.**MunichRetinaDevice**(*retina_key*, *spinnaker_link_id*, *position*, *label=None*, *polarity=None*, *board_address=None*)
Bases: pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex, spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.

AbstractProvidesOutgoingPartitionConstraints, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

DOWN_POLARITY = 'DOWN'

LEFT_RETINA = 'LEFT'

LEFT_RETINA_DISABLE = 69

LEFT_RETINA_ENABLE = 69

LEFT_RETINA_KEY_SET = 67

MANAGEMENT_BIT = 1024

MANAGEMENT_MASK = 4294965248

MERGED_POLARITY = 'MERGED'

RIGHT_RETINA = 'RIGHT'

RIGHT_RETINA_DISABLE = 70

RIGHT_RETINA_ENABLE = 70

RIGHT_RETINA_KEY_SET = 68

UP_POLARITY = 'UP'

default_parameters = {'board_address': None, 'label': 'MunichRetinaDevice', 'polarit

get_outgoing_partition_constraints (*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters partition (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

pause_stop_commands
The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands
The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

class spynnaker.pyNN.external_devices_models.**ThresholdTypeMulticastDeviceControl** (*device*)
Bases: *spynnaker.pyNN.models.neuron.threshold_types.abstract_threshold_type.AbstractThresholdType*

A threshold type that can send multicast keys with the value of membrane voltage as the payload

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (int) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (str) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (str) – The name of the variable

Return type bool

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

spynnaker.pyNN.model_binaries package

Module contents

This module contains no python code.

spynnaker.pyNN.models package

Subpackages

spynnaker.pyNN.models.abstract_models package

Submodules

spynnaker.pyNN.models.abstract_models.abstract_accepts_incoming_synapses module

class spynnaker.pyNN.models.abstract_models.abstract_accepts_incoming_synapses.**AbstractAcce**

Bases: object

Indicates an object that can be a post-vertex in a PyNN projection.

add_pre_run_connection_holder(*connection_holder*, *projection_edge*, *synapse_information*)
Add a connection holder to the vertex to be filled in when the connections are actually generated.

clear_connection_cache()
Clear the connection data stored in the vertex so far.

get_connections_from_machine(*transceiver*, *placement*, *edge*, *graph_mapper*, *routing_infos*, *synapse_information*, *machine_time_step*, *using_extra_monitor_cores*, *placements=None*, *monitor_api=None*, *monitor_placement=None*, *monitor_cores=None*, *handle_time_out_configuration=True*, *fixed_routes=None*)
Get the connections from the machine post-run.

get_maximum_delay_supported_in_ms(*machine_time_step*)
Get the maximum delay supported by this vertex.

get_synapse_id_by_target
Get the ID of a synapse given the name.

Parameters **target** (*str*) – The name of the synapse

Return type int

set_synapse_dynamics(*synapse_dynamics*)
Set the synapse dynamics of this vertex.

spynnaker.pyNN.models.abstract_models.abstract_contains_units module

class spynnaker.pyNN.models.abstract_models.abstract_contains_units.**AbstractContainsUnits**

Bases: object

get_units(*variable*)
Get units for a given variable

Parameters **variable** – the variable to find units from

Returns the units as a string.

spynnaker.pyNN.models.abstract_models.abstract_filterable_edge module

class spynnaker.pyNN.models.abstract_models.abstract_filterable_edge.**AbstractFilterableEdge**

Bases: object

An edge that can be filtered

filter_edge(*graph_mapper*)

Determine if this edge should be filtered out

Parameters **graph_mapper** – the mapper between graphs

Returns True if the edge should be filtered

Return type bool

spynnaker.pyNN.models.abstract_models.abstract_population_initializable module

class spynnaker.pyNN.models.abstract_models.abstract_population_initializable.**AbstractPopul**

Bases: object

Indicates that this object has properties that can be initialised by a PyNN Population

get_initial_value(*variable*, *selector=None*)

Gets the value for any variable whose in initialize_parameters.keys

Should return the current value not the default one.

Must support the variable as listed in initialize_parameters.keys, ideally also with *_init* removed or added.

Parameters

- **variable** (*str*) – variable name with our without _init
- **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

Returns A list or an Object which act like a list

get_initial_values(*selector=None*)

A dict containing the initial values of the state variables.

Parameters **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

initial_values

A dict containing the initial values of the state variables.

initialize(*variable*, *value*)

Set the initial value of one of the state variables of the neurons in this population.

initialize_parameters

List the parameters that are initializable.

If "foo" is initializable there should be a setter initialize_foo and a getter property foo_init

Returns list of property names

set_initial_value(*variable*, *value*, *selector=None*)

Sets the value for any variable whose in initialize_parameters.keys

Must support the variable as listed in initialize_parameters.keys, ideally also with *_init* removed or added

Parameters

- **variable** (*str*) – variable name with our without *_init*
- **value** – New value for the variable
- **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

Returns A list or an Object which act like a list

spynnaker.pyNN.models.abstract_models.abstract_population_settable module

class spynnaker.pyNN.models.abstract_models.abstract_population_settable.**AbstractPopulatio**

Bases: *spynnaker.pyNN.models.abstract_models.abstract_settable.AbstractSettable*

Indicates that some properties of this object can be accessed from the PyNN population set and get methods.

get_value_by_selector (*selector*, *key*)

Gets the value for a particular key but only for the selected subset.

Parameters

- **selector** – See RangedList.get_value_by_selector as this is just a pass through method
- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

n_atoms

" See ApplicationVertex.n_atoms

set_value_by_selector (*selector*, *key*, *value*)

Sets the value for a particular key but only for the selected subset.

Parameters

- **selector** – See RangedList.set_value_by_selector as this is just a pass through method
- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

spynnaker.pyNN.models.abstract_models.abstract_read_parameters_before_set module

class spynnaker.pyNN.models.abstract_models.abstract_read_parameters_before_set.**AbstractRea**

Bases: object

A vertex whose parameters must be read before any can be set

read_parameters_from_machine (*transceiver*, *placement*, *vertex_slice*)

Read the parameters from the machine before any are changed

Parameters

- **transceiver** – the SpinnMan interface
- **placement** – the placement of a vertex
- **vertex_slice** – the slice of atoms for this vertex

spynnaker.pyNN.models.abstract_models.abstract_settable module

class spynnaker.pyNN.models.abstract_models.abstract_settable.**AbstractSettable**
Bases: object

Indicates that some properties of this object can be accessed from the PyNN population set and get methods

get_value(*key*)
Get a property

set_value(*key*, *value*)
Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

spynnaker.pyNN.models.abstract_models.abstract_weight_updatable module

class spynnaker.pyNN.models.abstract_models.abstract_weight_updatable.**AbstractWeightUpdatab**
Bases: object

An object the weight of which can be updated

update_weight(*graph_mapper*)
Update the weight

Module contents

class spynnaker.pyNN.models.abstract_models.**AbstractAcceptsIncomingSynapses**
Bases: object

Indicates an object that can be a post-vertex in a PyNN projection.

add_pre_run_connection_holder(*connection_holder*, *projection_edge*, *synapse_information*)
Add a connection holder to the vertex to be filled in when the connections are actually generated.

clear_connection_cache()
Clear the connection data stored in the vertex so far.

get_connections_from_machine(*transceiver*, *placement*, *edge*, *graph_mapper*, *routing_infos*, *synapse_information*, *machine_time_step*, *using_extra_monitor_cores*, *placements=None*, *monitor_api=None*, *monitor_placement=None*, *monitor_cores=None*, *handle_time_out_configuration=True*, *fixed_routes=None*)
Get the connections from the machine post-run.

get_maximum_delay_supported_in_ms(*machine_time_step*)
Get the maximum delay supported by this vertex.

get_synapse_id_by_target
Get the ID of a synapse given the name.

Parameters **target** (str) – The name of the synapse

Return type int

set_synapse_dynamics(*synapse_dynamics*)
Set the synapse dynamics of this vertex.

class spynnaker.pyNN.models.abstract_models.**AbstractContainsUnits**
Bases: object

get_units(*variable*)
Get units for a given variable

Parameters **variable** – the variable to find units from

Returns the units as a string.

class spynnaker.pyNN.models.abstract_models.**AbstractFilterableEdge**
Bases: object

An edge that can be filtered

filter_edge(*graph_mapper*)
Determine if this edge should be filtered out

Parameters **graph_mapper** – the mapper between graphs

Returns True if the edge should be filtered

Return type bool

class spynnaker.pyNN.models.abstract_models.**AbstractPopulationInitializable**
Bases: object

Indicates that this object has properties that can be initialised by a PyNN Population

get_initial_value(*variable*, *selector=None*)
Gets the value for any variable whose in initialize_parameters.keys

Should return the current value not the default one.

Must support the variable as listed in initialize_parameters.keys, ideally also with *_init* removed or added.

Parameters

- **variable** (str) – variable name with our without _init
- **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

Returns A list or an Object which act like a list

get_initial_values(*selector=None*)
A dict containing the initial values of the state variables.

Parameters **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

initial_values
A dict containing the initial values of the state variables.

initialize(*variable*, *value*)
Set the initial value of one of the state variables of the neurons in this population.

initialize_parameters
List the parameters that are initializable.

If "foo" is initializable there should be a setter initialize_foo and a getter property foo_init

Returns list of property names

set_initial_value (*variable*, *value*, *selector=None*)
Sets the value for any variable whose in initialize_parameters.keys

Must support the variable as listed in initialize_parameters.keys, ideally also with *_init* removed or added

Parameters

- **variable** (`str`) – variable name with our without *_init*
- **value** – New value for the variable
- **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

Returns A list or an Object which act like a list

class `spynnaker.pyNN.models.abstract_models.`**AbstractPopulationSettable**
Bases: *spynnaker.pyNN.models.abstract_models.abstract_settable.AbstractSettable*

Indicates that some properties of this object can be accessed from the PyNN population set and get methods.

get_value_by_selector (*selector*, *key*)
Gets the value for a particular key but only for the selected subset.

Parameters

- **selector** – See RangedList.get_value_by_selector as this is just a pass through method
- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

n_atoms
" See ApplicationVertex.n_atoms

set_value_by_selector (*selector*, *key*, *value*)
Sets the value for a particular key but only for the selected subset.

Parameters

- **selector** – See RangedList.set_value_by_selector as this is just a pass through method
- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

class `spynnaker.pyNN.models.abstract_models.`**AbstractReadParametersBeforeSet**
Bases: `object`

A vertex whose parameters must be read before any can be set

read_parameters_from_machine (*transceiver*, *placement*, *vertex_slice*)
Read the parameters from the machine before any are changed

Parameters

- **transceiver** – the SpinnMan interface
- **placement** – the placement of a vertex
- **vertex_slice** – the slice of atoms for this vertex

class `spynnaker.pyNN.models.abstract_models.`**AbstractSettable**
Bases: `object`

Indicates that some properties of this object can be accessed from the PyNN population set and get methods

get_value (*key*)
Get a property

set_value (*key*, *value*)
Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

class spynnaker.pyNN.models.abstract_models.**AbstractWeightUpdatable**
Bases: object

An object the weight of which can be updated

update_weight (*graph_mapper*)
Update the weight

spynnaker.pyNN.models.common package

Submodules

spynnaker.pyNN.models.common.abstract_neuron_recordable module

class spynnaker.pyNN.models.common.abstract_neuron_recordable.**AbstractNeuronRecordable**
Bases: object

Indicates that a variable (e.g., membrane voltage) can be recorded from this object

clear_recording (*variable*, *buffer_manager*, *placements*, *graph_mapper*)
Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

get_data (*variable*, *n_machine_time_steps*, *placements*, *graph_mapper*, *buffer_manager*, *machine_time_step*)
Get the recorded data

Parameters

- **variable** –
- **n_machine_time_steps** –
- **placements** –
- **graph_mapper** –
- **buffer_manager** –
- **machine_time_step** –

Returns

get_neuron_sampling_interval (*variable*)
Returns the current sampling interval for this variable

Parameters **variable** – PyNN name of the variable

Returns Sampling interval in micro seconds

get_recordable_variables ()
Returns a list of the variables this models is expected to collect

is_recording (*variable*)
Determines if variable is being recorded

Returns True if variable are being recorded, False otherwise

Return type bool

set_recording (*variable*, *new_state=True*, *sampling_interval=None*, *indexes=None*)
Sets variable to being recorded

spynnaker.pyNN.models.common.abstract_spike_recordable module

class spynnaker.pyNN.models.common.abstract_spike_recordable.**AbstractSpikeRecordable**
Bases: `object`

Indicates that spikes can be recorded from this object

clear_spike_recording (*buffer_manager*, *placements*, *graph_mapper*)
Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

get_spikes (*placements*, *graph_mapper*, *buffer_manager*, *machine_time_step*)
Get the recorded spikes from the object

Parameters

- **placements** – the placements object
- **graph_mapper** – the graph mapper object
- **buffer_manager** – the buffer manager object
- **machine_time_step** – the time step of the simulation

Returns A numpy array of 2-element arrays of (neuron_id, time) ordered by time

get_spikes_sampling_interval ()
Return the current sampling interval for spikes

Returns Sampling interval in micro seconds

is_recording_spikes ()
Determine if spikes are being recorded

Returns True if spikes are being recorded, False otherwise

Return type bool

set_recording_spikes(*new_state=True*, *sampling_interval=None*, *indexes=None*)
Set spikes to being recorded. If new_state is false all other parameters are ignored.

Parameters

- **new_state** (*bool*) – Set if the spikes are recording or not
- **sampling_interval** – The interval at which spikes are recorded. Must be a whole multiple of the timestep None will be taken as the timestep
- **indexes** – The indexes of the neurons that will record spikes. If None the assumption is all neurons are recording

spynnaker.pyNN.models.common.eieio_spike_recorder module

class spynnaker.pyNN.models.common.eieio_spike_recorder.**EIEIOSpikeRecorder**
Bases: object

Records spikes using EIEIO format

get_dtcm_usage_in_bytes()

get_n_cpu_cycles(*n_neurons*)

get_spikes(*label*, *buffer_manager*, *region*, *placements*, *graph_mapper*, *application_vertex*, *base_key_function*, *machine_time_step*)

record

set_recording(*new_state*, *sampling_interval=None*)

spynnaker.pyNN.models.common.multi_spike_recorder module

class spynnaker.pyNN.models.common.multi_spike_recorder.**MultiSpikeRecorder**
Bases: object

get_dtcm_usage_in_bytes()

get_n_cpu_cycles(*n_neurons*)

get_sdram_usage_in_bytes(*n_neurons*, *spikes_per_timestep*)

get_spikes(*label*, *buffer_manager*, *region*, *placements*, *graph_mapper*, *application_vertex*, *machine_time_step*)

record

spynnaker.pyNN.models.common.neuron_recorder module

class spynnaker.pyNN.models.common.neuron_recorder.**NeuronRecorder**(*allowed_variables*, *n_neurons*)
Bases: object

MAX_RATE = 4294967295

N_BYTES_FOR_TIMESTAMP = 4

N_BYTES_PER_INDEX = 1

N_BYTES_PER_POINTER = 4

N_BYTES_PER_RATE = 4

N_BYTES_PER_SIZE = 4

N_BYTES_PER_VALUE = 4

N_BYTES_PER_WORD = 4

N_CPU_CYCLES_PER_NEURON = 8

SARK_BLOCK_SIZE = 8

check_indexes (*indexes*)

get_buffered_sdram (*variable*, *vertex_slice*, *n_machine_time_steps*)
Returns the SDRAM used for this may timesteps

If required the total is rounded up so the space will always fit

Parameters

- **variable** – The
- **vertex_slice** –

Returns

get_buffered_sdram_per_record (*variable*, *vertex_slice*)
Return the SDRAM used per record

Parameters

- **variable** –
- **vertex_slice** –

Returns

get_buffered_sdram_per_timestep (*variable*, *vertex_slice*)
Return the SDRAM used per timestep.

In the case where sampling is used it returns the average for recording and none recording based on the recording rate

Parameters

- **variable** –
- **vertex_slice** –

Returns

get_data (*vertex_slice*)

get_dtcm_usage_in_bytes (*vertex_slice*)

get_global_parameters (*vertex_slice*)

get_index_parameters (*vertex_slice*)

get_matrix_data (*label*, *buffer_manager*, *region*, *placements*, *graph_mapper*, *application_vertex*, *variable*, *n_machine_time_steps*)
Read a uint32 mapped to time and neuron IDs from the SpiNNaker machine.

Parameters

- **label** – vertex label
- **buffer_manager** – the manager for buffered data
- **region** – the DSG region ID used for this data

- **placements** – the placements object
- **graph_mapper** – the mapping between application and machine vertices
- **application_vertex** –
- **variable** (*str*) – PyNN name for the variable (V, gsy_inh etc.)
- **n_machine_time_steps** –

Returns

get_n_cpu_cycles(*n_neurons*)

get_neuron_sampling_interval(*variable*)
Return the current sampling interval for this variable

Parameters variable – PyNN name of the variable

Returns Sampling interval in micro seconds

get_recordable_variables()

get_sampling_overflow_sdram(*vertex_slice*)
Get the extra SDRAM that should be reserved if using per_timestep

This is the extra that must be reserved if per_timestep is an average rather than fixed for every timestep.

When sampling the average * time_steps may not be quite enough. This returns the extra space in the worst case where time_steps is a multiple of sampling rate + 1, and recording is done in the first and last time_step

Parameters vertex_slice –

Returns Highest possible overflow needed

get_sdram_usage_in_bytes(*vertex_slice*)

get_spikes(*label*, *buffer_manager*, *region*, *placements*, *graph_mapper*, *application_vertex*, *machine_time_step*)

get_variable_sdram_usage(*vertex_slice*)

is_recording(*variable*)

recorded_region_ids

recording_variables

set_recording(*variable*, *new_state*, *sampling_interval=None*, *indexes=None*)

spynnaker.pyNN.models.common.recording_utils module

spynnaker.pyNN.models.common.recording_utils.**get_buffer_sizes**(*buffer_max*, *space_needed*, *enable_buffered_recording*)

spynnaker.pyNN.models.common.recording_utils.**get_data**(*transceiver*, *placement*, *region*, *region_size*)
Get the recorded data from a region

spynnaker.pyNN.models.common.recording_utils.**get_recording_region_size_in_bytes**(*n_machine_tim*
bytes_per_time

Get the size of a recording region in bytes spynnaker.pyNN.models.common.recording_utils.**make_missing_string**(*missing*)

spynnaker.pyNN.models.common.recording_utils.**needs_buffering**(*buffer_max*, *space_needed*, *enable_buffered_recording*)

spynnaker.pyNN.models.common.recording_utils.**pull_off_cached_lists**(*no_loads*, *cache_file*)

Extracts numpy based data from a file

Parameters

- **no_loads** – the number of numpy elements in the file
- **cache_file** – the file to extract from

Returns The extracted data

spynnaker.pyNN.models.common.simple_population_settable module

class spynnaker.pyNN.models.common.simple_population_settable.**SimplePopulationSettable**

Bases: *spynnaker.pyNN.models.abstract_models.abstract_population_settable.AbstractPopulationSettable*

An object all of whose properties can be accessed from a PyNN Population i.e. no properties are hidden

get_value(*key*)

Get a property

set_value(*key*, *value*)

Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

Module contents

class spynnaker.pyNN.models.common.**AbstractNeuronRecordable**

Bases: object

Indicates that a variable (e.g., membrane voltage) can be recorded from this object

clear_recording(*variable*, *buffer_manager*, *placements*, *graph_mapper*)

Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

get_data(*variable*, *n_machine_time_steps*, *placements*, *graph_mapper*, *buffer_manager*, *machine_time_step*)

Get the recorded data

Parameters

- **variable** –
- **n_machine_time_steps** –
- **placements** –
- **graph_mapper** –
- **buffer_manager** –
- **machine_time_step** –

Returns

get_neuron_sampling_interval(*variable*)
Returns the current sampling interval for this variable

Parameters **variable** – PyNN name of the variable

Returns Sampling interval in micro seconds

get_recordable_variables()
Returns a list of the variables this models is expected to collect

is_recording(*variable*)
Determines if variable is being recorded

Returns True if variable are being recorded, False otherwise

Return type bool

set_recording(*variable*, *new_state=True*, *sampling_interval=None*, *indexes=None*)
Sets variable to being recorded

class spynnaker.pyNN.models.common.**AbstractSpikeRecordable**
Bases: object

Indicates that spikes can be recorded from this object

clear_spike_recording(*buffer_manager*, *placements*, *graph_mapper*)
Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

get_spikes(*placements*, *graph_mapper*, *buffer_manager*, *machine_time_step*)
Get the recorded spikes from the object

Parameters

- **placements** – the placements object
- **graph_mapper** – the graph mapper object
- **buffer_manager** – the buffer manager object
- **machine_time_step** – the time step of the simulation

Returns A numpy array of 2-element arrays of (neuron_id, time) ordered by time

get_spikes_sampling_interval()
Return the current sampling interval for spikes

Returns Sampling interval in micro seconds

is_recording_spikes()
Determine if spikes are being recorded

Returns True if spikes are being recorded, False otherwise

Return type bool

set_recording_spikes(*new_state=True*, *sampling_interval=None*, *indexes=None*)
Set spikes to being recorded. If new_state is false all other parameters are ignored.

Parameters

- **new_state** (*bool*) – Set if the spikes are recording or not
- **sampling_interval** – The interval at which spikes are recorded. Must be a whole multiple of the timestep None will be taken as the timestep
- **indexes** – The indexes of the neurons that will record spikes. If None the assumption is all neurons are recording

class spynnaker.pyNN.models.common.**EIEIOSpikeRecorder**
Bases: `object`

Records spikes using EIEIO format

get_dtcm_usage_in_bytes()

get_n_cpu_cycles(*n_neurons*)

get_spikes(*label*, *buffer_manager*, *region*, *placements*, *graph_mapper*, *application_vertex*, *base_key_function*, *machine_time_step*)

record

set_recording(*new_state*, *sampling_interval=None*)

class spynnaker.pyNN.models.common.**NeuronRecorder**(*allowed_variables*, *n_neurons*)
Bases: `object`

MAX_RATE = 4294967295

N_BYTES_FOR_TIMESTAMP = 4

N_BYTES_PER_INDEX = 1

N_BYTES_PER_POINTER = 4

N_BYTES_PER_RATE = 4

N_BYTES_PER_SIZE = 4

N_BYTES_PER_VALUE = 4

N_BYTES_PER_WORD = 4

N_CPU_CYCLES_PER_NEURON = 8

SARK_BLOCK_SIZE = 8

check_indexes(*indexes*)

get_buffered_sdram(*variable*, *vertex_slice*, *n_machine_time_steps*)
Returns the SDRAM used for this may timesteps

If required the total is rounded up so the space will always fit

Parameters

- **variable** – The
- **vertex_slice** –

Returns

get_buffered_sdram_per_record (*variable*, *vertex_slice*)
Return the SDRAM used per record

Parameters

- **variable** –
- **vertex_slice** –

Returns

get_buffered_sdram_per_timestep (*variable*, *vertex_slice*)
Return the SDRAM used per timestep.

In the case where sampling is used it returns the average for recording and none recording based on the recording rate

Parameters

- **variable** –
- **vertex_slice** –

Returns

get_data (*vertex_slice*)

get_dtcm_usage_in_bytes (*vertex_slice*)

get_global_parameters (*vertex_slice*)

get_index_parameters (*vertex_slice*)

get_matrix_data (*label*, *buffer_manager*, *region*, *placements*, *graph_mapper*, *application_vertex*, *variable*, *n_machine_time_steps*)
Read a uint32 mapped to time and neuron IDs from the SpiNNaker machine.

Parameters

- **label** – vertex label
- **buffer_manager** – the manager for buffered data
- **region** – the DSG region ID used for this data
- **placements** – the placements object
- **graph_mapper** – the mapping between application and machine vertices
- **application_vertex** –
- **variable** (*str*) – PyNN name for the variable (V, gsy_inh etc.)
- **n_machine_time_steps** –

Returns

get_n_cpu_cycles (*n_neurons*)

get_neuron_sampling_interval (*variable*)
Return the current sampling interval for this variable

Parameters **variable** – PyNN name of the variable

Returns Sampling interval in micro seconds

get_recordable_variables()

get_sampling_overflow_sdram(*vertex_slice*)
Get the extra SDRAM that should be reserved if using per_timestep

This is the extra that must be reserved if per_timestep is an average rather than fixed for every timestep.

When sampling the average * time_steps may not be quite enough. This returns the extra space in the worst case where time_steps is a multiple of sampling rate + 1, and recording is done in the first and last time_step

Parameters **vertex_slice** –

Returns Highest possible overflow needed

get_sdram_usage_in_bytes(*vertex_slice*)

get_spikes(*label*, *buffer_manager*, *region*, *placements*, *graph_mapper*, *application_vertex*, *machine_time_step*)

get_variable_sdram_usage(*vertex_slice*)

is_recording(*variable*)

recorded_region_ids

recording_variables

set_recording(*variable*, *new_state*, *sampling_interval=None*, *indexes=None*)

class spynnaker.pyNN.models.common.**MultiSpikeRecorder**
Bases: object

get_dtcm_usage_in_bytes()

get_n_cpu_cycles(*n_neurons*)

get_sdram_usage_in_bytes(*n_neurons*, *spikes_per_timestep*)

get_spikes(*label*, *buffer_manager*, *region*, *placements*, *graph_mapper*, *application_vertex*, *machine_time_step*)

record

class spynnaker.pyNN.models.common.**SimplePopulationSettable**
Bases: *spynnaker.pyNN.models.abstract_models.abstract_population_settable.AbstractPopulationSettable*

An object all of whose properties can be accessed from a PyNN Population i.e. no properties are hidden

get_value(*key*)
Get a property

set_value(*key*, *value*)
Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

spynnaker.pyNN.models.common.**get_buffer_sizes**(*buffer_max*, *space_needed*, *enable_buffered_recording*)

`spynnaker.pyNN.models.common.`**`get_data`**(*transceiver*, *placement*, *region*, *region_size*)
Get the recorded data from a region

`spynnaker.pyNN.models.common.`**`needs_buffering`**(*buffer_max*, *space_needed*, *enable_buffered_recording*)

`spynnaker.pyNN.models.common.`**`get_recording_region_size_in_bytes`**(*n_machine_time_steps*, *bytes_per_timestep*)
Get the size of a recording region in bytes

`spynnaker.pyNN.models.common.`**`pull_off_cached_lists`**(*no_loads*, *cache_file*)
Extracts numpy based data from a file

Parameters

- **`no_loads`** – the number of numpy elements in the file
- **`cache_file`** – the file to extract from

Returns The extracted data

spynnaker.pyNN.models.neural_projections package

Subpackages

spynnaker.pyNN.models.neural_projections.connectors package

Submodules

spynnaker.pyNN.models.neural_projections.connectors.abstract_connector module

class `spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.`**`AbstractConne`**

Bases: `object`

Abstract class that all PyNN Connectors extend.

`NUMPY_SYNAPSES_DTYPE = [('source', 'uint32'), ('target', 'uint16'), ('weight', 'float6`

`create_synaptic_block`(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

`get_delay_maximum`(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

`get_delay_variance`(*delays*)
Get the variance of the delays.

`get_n_connections_from_pre_vertex_maximum`(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

`get_n_connections_to_post_vertex_maximum`()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_provenance_data()

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

get_weight_mean(*weights*)
Get the mean of the weights.

get_weight_variance(*weights*)
Get the variance of the weights.

post_population

pre_population

safe

set_projection_information(*pre_population*, *post_population*, *rng*, *machine_time_step*)

set_space(*space*)
Set the space object (allowed after instantiation).

Parameters **space** –

Returns

space

verbose

spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine module

class spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_ma

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector*

Indicates that the connectivity can be generated on the machine

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

gen_delay_params(*delays*, *pre_vertex_slice*, *post_vertex_slice*)
Get the parameters of the delay generator on the machine

Return type numpy array of uint32

gen_delay_params_size_in_bytes(*delays*)
The size of the delay parameters in bytes

Return type int

gen_delays_id(*delays*)
Get the id of the delay generator on the machine

Return type int

gen_weight_params_size_in_bytes(*weights*)
The size of the weight parameters in bytes

Return type int

gen_weights_id(*weights*)
Get the id of the weight generator on the machine

Return type int

gen_weights_params(*weights*, *pre_vertex_slice*, *post_vertex_slice*)
Get the parameters of the weight generator on the machine

Return type numpy array of uint32

generate_on_machine(*weights*, *delays*)
Determine if this instance can generate on the machine.

Default implementation returns True if the weights and delays can be generated on the machine

Return type bool

class spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_ma
Bases: enum.Enum

An enumeration.

ALL_TO_ALL_CONNECTOR = 1

FIXED_NUMBER_POST_CONNECTOR = 5

FIXED_NUMBER_PRE_CONNECTOR = 4

FIXED_PROBABILITY_CONNECTOR = 2

FIXED_TOTAL_NUMBER_CONNECTOR = 3

KERNEL_CONNECTOR = 6

ONE_TO_ONE_CONNECTOR = 0

spynnaker.pyNN.models.neural_projections.connectors.all_to_all_connector module

class spynnaker.pyNN.models.neural_projections.connectors.all_to_all_connector.**AllToAllConn**

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

Connects all cells in the presynaptic population to all cells in the postsynaptic population.

Parameters **allow_self_connections** (*bool*) – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.

allow_self_connections

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

spynnaker.pyNN.models.neural_projections.connectors.array_connector module

class spynnaker.pyNN.models.neural_projections.connectors.array_connector.**ArrayConnector**(*ar sa ca ba ve ba*

Bases: `spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector`

Make connections using an array of integers based on the IDs of the neurons in the pre- and post-populations.

Parameters array – An explicit boolean matrix that specifies the connections between the pre- and post-populations (see PyNN documentation)

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons

in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

spynnaker.pyNN.models.neural_projections.connectors.csa_connector module

class spynnaker.pyNN.models.neural_projections.connectors.csa_connector.**CSAConnector**(*cset*, *safe=Tr* *call-back=N* *ver-bose=Fa*

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector*

Make connections using a Connection Set Algebra (Djurfeldt 2012) description between the neurons in the pre- and post-populations. If you get TypeError in Python 3 see: https://github.com/INCF/csa/issues/10

Parameters **cset** (*'?'*) – A description of the connection set between populations

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

show_connection_set()

spynnaker.pyNN.models.neural_projections.connectors.distance_dependent_probability_connector module

class spynnaker.pyNN.models.neural_projections.connectors.distance_dependent_probability_co

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector*

Make connections using a distribution which varies with distance.

Parameters

- **d_expression** (*bool*) – the right-hand side of a valid python expression for probability, involving 'd', e.g. "exp(-abs(d))", or "d<3", that can be parsed by eval(), that computes the distance dependent distribution.
- **allow_self_connections** – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.
- **space** (*pyNN.Space*) – a Space object, needed if you wish to specify distance-dependent weights or delays.
- **n_connections** (*int or None*) – The number of efferent synaptic connections per neuron.

allow_self_connections

create_synaptic_block(*weights, delays, pre_slices, pre_slice_index, post_slices, post_slice_index, pre_vertex_slice, post_vertex_slice, synapse_type*)
Create a synaptic block from the data.

d_expression

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays, post_vertex_slice, min_delay=None, max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

set_projection_information(*pre_population, post_population, rng, machine_time_step*)

spynnaker.pyNN.models.neural_projections.connectors.fixed_number_post_connector module

class spynnaker.pyNN.models.neural_projections.connectors.fixed_number_post_connector.**Fixed**

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

Connects a fixed number of post-synaptic neurons selected at random, to all pre-synaptic neurons.

Parameters

- **n** (*int*) – number of random post-synaptic neurons connected to pre-neurons.
- **allow_self_connections** (*bool*) – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.
- **with_replacement** (*bool*) – this flag determines how the random selection of post-synaptic neurons is performed; if true, then every post-synaptic neuron can be chosen on each occasion, and so multiple connections between neuron pairs are possible; if false, then once a post-synaptic neuron has been connected to a pre-neuron, it can't be connected again.

allow_self_connections

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

set_projection_information(*pre_population*, *post_population*, *rng*, *machine_time_step*)

spynnaker.pyNN.models.neural_projections.connectors.fixed_number_pre_connector module

class spynnaker.pyNN.models.neural_projections.connectors.fixed_number_pre_connector.**FixedN**

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

Connects a fixed number of pre-synaptic neurons selected at random, to all post-synaptic neurons.

Parameters

- **n** (*int*) – number of random pre-synaptic neurons connected to output
- **allow_self_connections** (*bool*) – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.
- **with_replacement** (*bool*) – this flag determines how the random selection of pre-synaptic neurons is performed; if true, then every pre-synaptic neuron can be chosen on each occasion, and so multiple connections between neuron pairs are possible; if false, then once a pre-synaptic neuron has been connected to a post-neuron, it can't be connected again.

allow_self_connections

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons

in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

set_projection_information(*pre_population*, *post_population*, *rng*, *machine_time_step*)

spynnaker.pyNN.models.neural_projections.connectors.fixed_probability_connector module

class spynnaker.pyNN.models.neural_projections.connectors.fixed_probability_connector.**Fixed**

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

For each pair of pre-post cells, the connection probability is constant.

Parameters

- **p_connect** (*float*) – a float between zero and one. Each potential connection is created with this probability.
- **allow_self_connections** (*bool*) – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.
- **space** (*pyNN.Space*) – a Space object, needed if you wish to specify distance-dependent weights or delays - not implemented

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

spynnaker.pyNN.models.neural_projections.connectors.from_list_connector module

class spynnaker.pyNN.models.neural_projections.connectors.from_list_connector.**FromListConne**

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector*

Make connections according to a list.

Param conn_list: a list of tuples, one tuple for each connection. Each tuple should contain at least:

```
(pre_idx, post_idx)
```

where pre_idx is the index (i.e. order in the Population, not the ID) of the presynaptic neuron, and post_idx is the index of the postsynaptic neuron.

Additional items per synapse are acceptable but all synapses should have the same number of items.

column_names

conn_list

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_delay_variance(*delays*)
Get the variance of the delays.

get_extra_parameter_names()
Getter for the names of the extra parameters

get_extra_parameters()
Getter for the extra parameters.

Returns The extra parameters

get_n_connections(*pre_slices*, *post_slices*, *pre_hi*, *post_hi*)

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

get_weight_mean(*weights*)
Get the mean of the weights.

get_weight_variance(*weights*)
Get the variance of the weights.

spynnaker.pyNN.models.neural_projections.connectors.index_based_probability_connector module

class spynnaker.pyNN.models.neural_projections.connectors.index_based_probability_connecto

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector*

Make connections using a probability distribution which varies dependent upon the indices of the pre- and post-populations.

Parameters

- **index_expression** (*string*) – the right-hand side of a valid python expression for probability, involving the indices of the pre and post populations, that can be parsed by eval(), that computes a probability dist.
- **allow_self_connections** (*bool*) – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.

allow_self_connections

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons

in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

index_expression

spynnaker.pyNN.models.neural_projections.connectors.kernel_connector module

class spynnaker.pyNN.models.neural_projections.connectors.kernel_connector.**ConvolutionKerne**
Bases: numpy.ndarray

class spynnaker.pyNN.models.neural_projections.connectors.kernel_connector.**KernelConnector**

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

Where the pre- and post-synaptic populations are considered as a 2D array. Connect every post(row, col) neuron to many pre(row, col, kernel) through a (kernel) set of weights and/or delays.

TODO: should these include allow_self_connections and with_replacement?

Parameters

- **shape_pre** – 2D shape of the pre population (rows/height, cols/width, usually the input image shape)
- **shape_post** – 2D shape of the post population (rows/height, cols/width)
- **shape_kernel** – 2D shape of the kernel (rows/height, cols/width)
- **(optional)** (*pre/post_start_coords*) – 2D matrix of size shape_kernel describing the weights
- **(optional)** – 2D matrix of size shape_kernel describing the delays
- **(optional)** – 2D shape of common coordinate system (for both pre and post, usually the input image sizes)
- **(optional)** – Sampling steps/jumps for pre/post pop <=> (startX, endX, _stepX_) None or 2-item array

- **(optional)** – Starting row/col for pre/post sampling <=> (_startX_, endX, stepX) None or 2-item array

compute_statistics (*weights*, *delays*, *pre_vertex_slice*, *post_vertex_slice*)

create_synaptic_block (*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params (*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

gen_delay_params (*delays*, *pre_vertex_slice*, *post_vertex_slice*)
Get the parameters of the delay generator on the machine

Return type numpy array of uint32

gen_delay_params_size_in_bytes (*delays*)
The size of the delay parameters in bytes

Return type int

gen_delays_id (*delays*)
Get the id of the delay generator on the machine

Return type int

gen_weight_params_size_in_bytes (*weights*)
The size of the weight parameters in bytes

Return type int

gen_weights_id (*weights*)
Get the id of the weight generator on the machine

Return type int

gen_weights_params (*weights*, *pre_vertex_slice*, *post_vertex_slice*)
Get the parameters of the weight generator on the machine

Return type numpy array of uint32

generate_on_machine
Determine if this instance can generate on the machine.

Default implementation returns True if the weights and delays can be generated on the machine

Return type bool

get_delay_maximum (*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_kernel_vals (*vals*)

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

map_to_pre_coords(*post_r*, *post_c*)

post_as_pre(*post_vertex_slice*)

pre_as_post(*coords*)

to_post_coords(*post_vertex_slice*)

spynnaker.pyNN.models.neural_projections.connectors.kernel_connector.**shape2word**(*sw*, *sh*)

spynnaker.pyNN.models.neural_projections.connectors.multapse_connector module

class spynnaker.pyNN.models.neural_projections.connectors.multapse_connector.**MultapseConne**

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

Create a multapse connector. The size of the source and destination populations are obtained when the projection is connected. The number of synapses is specified. when instantiated, the required number of synapses is created by selecting at random from the source and target populations with replacement. Uniform selection probability is assumed.

Parameters

- **num_synapses** (*int*) – This is the total number of synapses in the connection.
- **allow_self_connections** (*bool*) – Allow a neuron to connect to itself or not.
- **with_replacement** (*bool*) – When selecting, allow a neuron to be re-selected or not.

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_rng_next(*num_synapses*, *prob_connect*)
Get the required RNGs

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

spynnaker.pyNN.models.neural_projections.connectors.one_to_one_connector module

class spynnaker.pyNN.models.neural_projections.connectors.one_to_one_connector.**OneToOneConn**

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

Where the pre- and postsynaptic populations have the same size, connect cell i in the presynaptic pynn_population.py to cell i in the postsynaptic pynn_population.py for all i.

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

spynnaker.pyNN.models.neural_projections.connectors.small_world_connector module

class spynnaker.pyNN.models.neural_projections.connectors.small_world_connector.**SmallWorldC**

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector*

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

set_projection_information(*pre_population*, *post_population*, *rng*, *machine_time_step*)

Module contents

class spynnaker.pyNN.models.neural_projections.connectors.**AbstractConnector**(*safe=True*, *verbose=False*, *rng=None*)

Bases: object

Abstract class that all PyNN Connectors extend.

NUMPY_SYNAPSES_DTYPE = [('source', 'uint32'), ('target', 'uint16'), ('weight', 'float6

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_delay_variance (*delays*)
Get the variance of the delays.

get_n_connections_from_pre_vertex_maximum (*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum ()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_provenance_data ()

get_weight_maximum (*weights*)
Get the maximum of the weights for this connection.

get_weight_mean (*weights*)
Get the mean of the weights.

get_weight_variance (*weights*)
Get the variance of the weights.

post_population

pre_population

safe

set_projection_information (*pre_population*, *post_population*, *rng*, *machine_time_step*)

set_space (*space*)
Set the space object (allowed after instantiation).

Parameters **space** –

Returns

space

verbose

class spynnaker.pyNN.models.neural_projections.connectors.**AbstractGenerateConnectorOnMachi**

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector*

Indicates that the connectivity can be generated on the machine

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params (*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

gen_delay_params (*delays*, *pre_vertex_slice*, *post_vertex_slice*)
Get the parameters of the delay generator on the machine

Return type numpy array of uint32

gen_delay_params_size_in_bytes (*delays*)
The size of the delay parameters in bytes

Return type int

gen_delays_id (*delays*)
Get the id of the delay generator on the machine

Return type int

gen_weight_params_size_in_bytes (*weights*)
The size of the weight parameters in bytes

Return type int

gen_weights_id (*weights*)
Get the id of the weight generator on the machine

Return type int

gen_weights_params (*weights*, *pre_vertex_slice*, *post_vertex_slice*)
Get the parameters of the weight generator on the machine

Return type numpy array of uint32

generate_on_machine (*weights*, *delays*)
Determine if this instance can generate on the machine.

Default implementation returns True if the weights and delays can be generated on the machine

Return type bool

class spynnaker.pyNN.models.neural_projections.connectors.**AllToAllConnector** (*allow_self_connectio safe=True, verbose=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine

Connects all cells in the presynaptic population to all cells in the postsynaptic population.

Parameters allow_self_connections (bool) – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.

allow_self_connections

create_synaptic_block (*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params (*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

class spynnaker.pyNN.models.neural_projections.connectors.**ArrayConnector**(*array*, *safe=True*, *callback=None*, *verbose=False*)
Bases: `spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector`

Make connections using an array of integers based on the IDs of the neurons in the pre- and post-populations.

Parameters **array** – An explicit boolean matrix that specifies the connections between the pre- and post-populations (see PyNN documentation)

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

class spynnaker.pyNN.models.neural_projections.connectors.**CSAConnector**(*cset*, *safe=True*, *callback=None*, *verbose=False*)

Bases: `spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector`

Make connections using a Connection Set Algebra (Djurfeldt 2012) description between the neurons in the pre- and post-populations. If you get TypeError in Python 3 see: https://github.com/INCF/csa/issues/10

Parameters **cset** (`'?'`) – A description of the connection set between populations

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

show_connection_set()

class spynnaker.pyNN.models.neural_projections.connectors.**DistanceDependentProbabilityConne**

Bases: `spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector`

Make connections using a distribution which varies with distance.

Parameters

- **d_expression** (`bool`) – the right-hand side of a valid python expression for probability, involving 'd', e.g. "exp(-abs(d))", or "d<3", that can be parsed by eval(), that computes the distance dependent distribution.
- **allow_self_connections** – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.
- **space** (`pyNN.Space`) – a Space object, needed if you wish to specify distance-dependent weights or delays.
- **n_connections** (`int or None`) – The number of efferent synaptic connections per neuron.

allow_self_connections

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

d_expression

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

set_projection_information(*pre_population*, *post_population*, *rng*, *machine_time_step*)

class spynnaker.pyNN.models.neural_projections.connectors.**FixedNumberPostConnector**(*n*, *al-low_self_c* *with_repla* *safe=True*, *ver-bose=Fals* *rng=None*)

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

Connects a fixed number of post-synaptic neurons selected at random, to all pre-synaptic neurons.

Parameters

- **n** (*int*) – number of random post-synaptic neurons connected to pre-neurons.
- **allow_self_connections** (*bool*) – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.
- **with_replacement** (*bool*) – this flag determines how the random selection of post-synaptic neurons is performed; if true, then every post-synaptic neuron can be chosen on each occasion, and so multiple connections between neuron pairs are possible; if false, then once a post-synaptic neuron has been connected to a pre-neuron, it can't be connected again.

allow_self_connections

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

set_projection_information(*pre_population*, *post_population*, *rng*, *machine_time_step*)

class spynnaker.pyNN.models.neural_projections.connectors.**FixedNumberPreConnector**(*n*, *allow_self_connections*, *with_replacement*, *safe=True*, *verbose=False*, *rng=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine

Connects a fixed number of pre-synaptic neurons selected at random, to all post-synaptic neurons.

Parameters

- **n** (*int*) – number of random pre-synaptic neurons connected to output
- **allow_self_connections** (*bool*) – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.
- **with_replacement** (*bool*) – this flag determines how the random selection of pre-synaptic neurons is performed; if true, then every pre-synaptic neuron can be chosen on each occasion, and so multiple connections between neuron pairs are possible; if false, then once a pre-synaptic neuron has been connected to a post-neuron, it can't be connected again.

allow_self_connections

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

set_projection_information(*pre_population*, *post_population*, *rng*, *machine_time_step*)

class spynnaker.pyNN.models.neural_projections.connectors.**FixedProbabilityConnector**(*p_connec al- low_self_ safe=Tru ver- bose=Fal rng=None*

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

For each pair of pre-post cells, the connection probability is constant.

Parameters

- **p_connect** (*float*) – a float between zero and one. Each potential connection is created with this probability.
- **allow_self_connections** (*bool*) – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.
- **space** (*pyNN.Space*) – a Space object, needed if you wish to specify distance-dependent weights or delays - not implemented

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

class spynnaker.pyNN.models.neural_projections.connectors.**FromListConnector**(*conn_list*, *safe=True*, *verbose=False*, *column_names=None*)

Bases: `spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector`

Make connections according to a list.

Param conn_list: a list of tuples, one tuple for each connection. Each tuple should contain at least:

```
(pre_idx, post_idx)
```

where pre_idx is the index (i.e. order in the Population, not the ID) of the presynaptic neuron, and post_idx is the index of the postsynaptic neuron.

Additional items per synapse are acceptable but all synapses should have the same number of items.

column_names

conn_list

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_delay_variance(*delays*)
Get the variance of the delays.

get_extra_parameter_names()
Getter for the names of the extra parameters

get_extra_parameters()
Getter for the extra parameters.

Returns The extra parameters

get_n_connections(*pre_slices*, *post_slices*, *pre_hi*, *post_hi*)

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

get_weight_mean(*weights*)
Get the mean of the weights.

get_weight_variance(*weights*)
Get the variance of the weights.

class spynnaker.pyNN.models.neural_projections.connectors.**IndexBasedProbabilityConnector**(*in al lo rn sa ca bo ve bo*

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector*

Make connections using a probability distribution which varies dependent upon the indices of the pre- and post-populations.

Parameters

- **index_expression** (*string*) – the right-hand side of a valid python expression for probability, involving the indices of the pre and post populations, that can be parsed by eval(), that computes a probability dist.
- **allow_self_connections** (*bool*) – if the connector is used to connect a Population to itself, this flag determines whether a neuron is allowed to connect to itself, or only to other neurons in the Population.

allow_self_connections

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons

in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

index_expression

class spynnaker.pyNN.models.neural_projections.connectors.**MultapseConnector**(*num_synapses*, *allow_self_connections*, *with_replacement=Tr*, *safe=True*, *verbose=False*, *rng=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine

Create a multapse connector. The size of the source and destination populations are obtained when the projection is connected. The number of synapses is specified. when instantiated, the required number of synapses is created by selecting at random from the source and target populations with replacement. Uniform selection probability is assumed.

Parameters

- **num_synapses** (*int*) – This is the total number of synapses in the connection.
- **allow_self_connections** (*bool*) – Allow a neuron to connect to itself or not.
- **with_replacement** (*bool*) – When selecting, allow a neuron to be re-selected or not.

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_rng_next(*num_synapses*, *prob_connect*)
Get the required RNGs

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

class spynnaker.pyNN.models.neural_projections.connectors.**OneToOneConnector**(*random_number_clas safe=True*, *verbose=False*)

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

Where the pre- and postsynaptic populations have the same size, connect cell i in the presynaptic pynn_population.py to cell i in the postsynaptic pynn_population.py for all i.

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

class spynnaker.pyNN.models.neural_projections.connectors.**SmallWorldConnector**(*degree*, *rewiring*, *allow_self_connectio safe=True*, *verbose=False*, *n_connections=No*

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector*

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

set_projection_information(*pre_population*, *post_population*, *rng*, *machine_time_step*)

class spynnaker.pyNN.models.neural_projections.connectors.**KernelConnector**(*shape_pre*, *shape_post*, *shape_kernel*, *weight_kernel*, *delay_kernel*, *shape_common*, *pre_sample_steps*, *pre_start_coords*, *post_sample_steps*, *post_start_coords*, *safe*, *space*, *verbose*)

Bases: *spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine*

Where the pre- and post-synaptic populations are considered as a 2D array. Connect every post(row, col) neuron to many pre(row, col, kernel) through a (kernel) set of weights and/or delays.

TODO: should these include allow_self_connections and with_replacement?

Parameters

- **shape_pre** – 2D shape of the pre population (rows/height, cols/width, usually the input image shape)
- **shape_post** – 2D shape of the post population (rows/height, cols/width)
- **shape_kernel** – 2D shape of the kernel (rows/height, cols/width)
- **(optional)** (*pre/post_start_coords*) – 2D matrix of size shape_kernel describing the weights
- **(optional)** – 2D matrix of size shape_kernel describing the delays
- **(optional)** – 2D shape of common coordinate system (for both pre and post, usually the input image sizes)
- **(optional)** – Sampling steps/jumps for pre/post pop <=> (startX, endX, _stepX_) None or 2-item array
- **(optional)** – Starting row/col for pre/post sampling <=> (_startX_, endX, stepX) None or 2-item array

compute_statistics(*weights*, *delays*, *pre_vertex_slice*, *post_vertex_slice*)

create_synaptic_block(*weights*, *delays*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Create a synaptic block from the data.

gen_connector_id
Get the id of the connection generator on the machine

Return type int

gen_connector_params(*pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*)
Get the parameters of the on machine generation.

Return type numpy array of uint32

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

gen_delay_params(*delays*, *pre_vertex_slice*, *post_vertex_slice*)
Get the parameters of the delay generator on the machine

Return type numpy array of uint32

gen_delay_params_size_in_bytes(*delays*)
The size of the delay parameters in bytes

Return type int

gen_delays_id(*delays*)
Get the id of the delay generator on the machine

Return type int

gen_weight_params_size_in_bytes(*weights*)
The size of the weight parameters in bytes

Return type int

gen_weights_id(*weights*)
Get the id of the weight generator on the machine

Return type int

gen_weights_params(*weights*, *pre_vertex_slice*, *post_vertex_slice*)
Get the parameters of the weight generator on the machine

Return type numpy array of uint32

generate_on_machine
Determine if this instance can generate on the machine.

Default implementation returns True if the weights and delays can be generated on the machine

Return type bool

get_delay_maximum(*delays*)
Get the maximum delay specified by the user in ms, or None if unbounded.

get_kernel_vals(*vals*)

get_n_connections_from_pre_vertex_maximum(*delays*, *post_vertex_slice*, *min_delay=None*, *max_delay=None*)
Get the maximum number of connections between those from any neuron in the pre vertex to the neurons

in the post_vertex_slice, for connections with a delay between min_delay and max_delay (inclusive) if both specified (otherwise all connections).

get_n_connections_to_post_vertex_maximum()
Get the maximum number of connections between those to any neuron in the post vertex from neurons in the pre vertex.

get_weight_maximum(*weights*)
Get the maximum of the weights for this connection.

map_to_pre_coords(*post_r*, *post_c*)

post_as_pre(*post_vertex_slice*)

pre_as_post(*coords*)

to_post_coords(*post_vertex_slice*)

Submodules

spynnaker.pyNN.models.neural_projections.delay_afferent_application_edge module

class spynnaker.pyNN.models.neural_projections.delay_afferent_application_edge.**DelayAfferer**

Bases: pacman.model.graphs.application.application_edge.ApplicationEdge

create_machine_edge(*pre_vertex*, *post_vertex*, *label*)
Create a machine edge between two machine vertices

Parameters

- **pre_vertex** (pacman.model.graphs.machine.MachineVertex) – The machine vertex at the start of the edge
- **post_vertex** (pacman.model.graphs.machine.MachineVertex) – The machine vertex at the end of the edge
- **label** (str) – label of the edge

Returns The created machine edge

Return type pacman.model.graphs.machine.MachineEdge

spynnaker.pyNN.models.neural_projections.delay_afferent_machine_edge module

class spynnaker.pyNN.models.neural_projections.delay_afferent_machine_edge.**DelayAfferentMac**

Bases: pacman.model.graphs.machine.machine_edge.MachineEdge, *spynnaker.pyNN.models.abstract_models.abstract_filterable_edge.AbstractFilterableEdge*, *spynnaker.pyNN.models.abstract_models.abstract_weight_updatable.AbstractWeightUpdatable*

filter_edge(*graph_mapper*)
Determine if this edge should be filtered out

Parameters **graph_mapper** – the mapper between graphs

Returns True if the edge should be filtered

Return type bool

update_weight(*graph_mapper*)
Update the weight

spynnaker.pyNN.models.neural_projections.delayed_application_edge module

class spynnaker.pyNN.models.neural_projections.delayed_application_edge.**DelayedApplicationI**

Bases: pacman.model.graphs.application.application_edge.ApplicationEdge

add_synapse_information(*synapse_information*)

create_machine_edge(*pre_vertex*, *post_vertex*, *label*)
Create a machine edge between two machine vertices

Parameters

- **pre_vertex** (pacman.model.graphs.machine.MachineVertex) – The machine vertex at the start of the edge
- **post_vertex** (pacman.model.graphs.machine.MachineVertex) – The machine vertex at the end of the edge
- **label** (*str*) – label of the edge

Returns The created machine edge

Return type pacman.model.graphs.machine.MachineEdge

synapse_information

spynnaker.pyNN.models.neural_projections.delayed_machine_edge module

class spynnaker.pyNN.models.neural_projections.delayed_machine_edge.**DelayedMachineEdge**(*synap*
pre_v
*post_*
la-
bel=N
weigh

Bases: pacman.model.graphs.machine.machine_edge.MachineEdge, *spynnaker.pyNN.models.abstract_models.abstract_filterable_edge.AbstractFilterableEdge*

filter_edge(*graph_mapper*)
Determine if this edge should be filtered out

Parameters **graph_mapper** – the mapper between graphs

Returns True if the edge should be filtered

Return type bool

spynnaker.pyNN.models.neural_projections.projection_application_edge module

class spynnaker.pyNN.models.neural_projections.projection_application_edge.**ProjectionApplic**

Bases: pacman.model.graphs.application.application_edge.ApplicationEdge

An edge which terminates on an AbstractPopulationVertex.

add_synapse_information (*synapse_information*)

create_machine_edge (*pre_vertex*, *post_vertex*, *label*)
Create a machine edge between two machine vertices

Parameters

- **pre_vertex** (pacman.model.graphs.machine.MachineVertex) – The machine vertex at the start of the edge
- **post_vertex** (pacman.model.graphs.machine.MachineVertex) – The machine vertex at the end of the edge
- **label** (*str*) – label of the edge

Returns The created machine edge

Return type pacman.model.graphs.machine.MachineEdge

delay_edge

n_delay_stages

synapse_information

spynnaker.pyNN.models.neural_projections.projection_machine_edge module

class spynnaker.pyNN.models.neural_projections.projection_machine_edge.**ProjectionMachineEdg**

Bases: pacman.model.graphs.machine.machine_edge.MachineEdge, *spynnaker.pyNN.models.abstract_models.abstract_filterable_edge.AbstractFilterableEdge*, *spynnaker.pyNN.models.abstract_models.abstract_weight_updatable.AbstractWeightUpdatable*, spinn_front_end_common.interface.provenance.abstract_provides_local_provenance_data.AbstractProvidesLocalProvenanceData

filter_edge (*graph_mapper*)
Determine if this edge should be filtered out

Parameters **graph_mapper** – the mapper between graphs

Returns True if the edge should be filtered

Return type bool

get_local_provenance_data()
Get an iterable of provenance data items

Returns iterable of `ProvenanceDataItem`

synapse_information

update_weight(*graph_mapper*)
Update the weight

spynnaker.pyNN.models.neural_projections.synapse_information module

class spynnaker.pyNN.models.neural_projections.synapse_information.**SynapseInformation**(*connec synaps synaps weight de- lay=N*)

Bases: `object`

Contains the synapse information including the connector, synapse type and synapse dynamics

connector

delay

synapse_dynamics

synapse_type

weight

Module contents

class spynnaker.pyNN.models.neural_projections.**DelayAfferentApplicationEdge**(*prevertex, delayvertex, label=None*)

Bases: `pacman.model.graphs.application.application_edge.ApplicationEdge`

create_machine_edge(*pre_vertex, post_vertex, label*)
Create a machine edge between two machine vertices

Parameters

- **pre_vertex** (`pacman.model.graphs.machine.MachineVertex`) – The machine vertex at the start of the edge
- **post_vertex** (`pacman.model.graphs.machine.MachineVertex`) – The machine vertex at the end of the edge
- **label** (`str`) – label of the edge

Returns The created machine edge

Return type `pacman.model.graphs.machine.MachineEdge`

class spynnaker.pyNN.models.neural_projections.**DelayAfferentMachineEdge**(*pre_vertex*, *post_vertex*, *label*, *weight=1*)

Bases: pacman.model.graphs.machine.machine_edge.MachineEdge, *spynnaker.pyNN.models.abstract_models.abstract_filterable_edge.AbstractFilterableEdge*, *spynnaker.pyNN.models.abstract_models.abstract_weight_updatable.AbstractWeightUpdatable*

filter_edge(*graph_mapper*)
Determine if this edge should be filtered out

Parameters **graph_mapper** – the mapper between graphs

Returns True if the edge should be filtered

Return type bool

update_weight(*graph_mapper*)
Update the weight

class spynnaker.pyNN.models.neural_projections.**DelayedApplicationEdge**(*pre_vertex*, *post_vertex*, *synapse_information*, *label=None*)

Bases: pacman.model.graphs.application.application_edge.ApplicationEdge

add_synapse_information(*synapse_information*)

create_machine_edge(*pre_vertex*, *post_vertex*, *label*)
Create a machine edge between two machine vertices

Parameters

- **pre_vertex** (pacman.model.graphs.machine.MachineVertex) – The machine vertex at the start of the edge
- **post_vertex** (pacman.model.graphs.machine.MachineVertex) – The machine vertex at the end of the edge
- **label** (*str*) – label of the edge

Returns The created machine edge

Return type pacman.model.graphs.machine.MachineEdge

synapse_information

class spynnaker.pyNN.models.neural_projections.**DelayedMachineEdge**(*synapse_information*, *pre_vertex*, *post_vertex*, *label=None*, *weight=1*)

Bases: pacman.model.graphs.machine.machine_edge.MachineEdge, *spynnaker.pyNN.models.abstract_models.abstract_filterable_edge.AbstractFilterableEdge*

filter_edge(*graph_mapper*)
Determine if this edge should be filtered out

Parameters **graph_mapper** – the mapper between graphs

Returns True if the edge should be filtered

Return type bool

class spynnaker.pyNN.models.neural_projections.**ProjectionApplicationEdge**(*pre_vertex*, *post_vertex*, *synapse_information*, *label=None*)

Bases: pacman.model.graphs.application.application_edge.ApplicationEdge

An edge which terminates on an AbstractPopulationVertex.

add_synapse_information(*synapse_information*)

create_machine_edge(*pre_vertex*, *post_vertex*, *label*)

Create a machine edge between two machine vertices

Parameters

- **pre_vertex** (pacman.model.graphs.machine.MachineVertex) – The machine vertex at the start of the edge
- **post_vertex** (pacman.model.graphs.machine.MachineVertex) – The machine vertex at the end of the edge
- **label** (*str*) – label of the edge

Returns The created machine edge

Return type pacman.model.graphs.machine.MachineEdge

delay_edge

n_delay_stages

synapse_information

class spynnaker.pyNN.models.neural_projections.**ProjectionMachineEdge**(*synapse_information*, *pre_vertex*, *post_vertex*, *label=None*, *traffic_weight=1*)

Bases: pacman.model.graphs.machine.machine_edge.MachineEdge, *spynnaker.pyNN.models.abstract_models.abstract_filterable_edge.AbstractFilterableEdge*, *spynnaker.pyNN.models.abstract_models.abstract_weight_updatable.AbstractWeightUpdatable*, spinn_front_end_common.interface.provenance.abstract_provides_local_provenance_data.AbstractProvidesLocalProvenanceData

filter_edge(*graph_mapper*)

Determine if this edge should be filtered out

Parameters **graph_mapper** – the mapper between graphs

Returns True if the edge should be filtered

Return type bool

get_local_provenance_data()

Get an iterable of provenance data items

Returns iterable of ProvenanceDataItem

synapse_information

update_weight (*graph_mapper*)
Update the weight

class spynnaker.pyNN.models.neural_projections.**SynapseInformation** (*connector, synapse_dynamics, synapse_type, weight=None, delay=None*)

Bases: object

Contains the synapse information including the connector, synapse type and synapse dynamics

connector

delay

synapse_dynamics

synapse_type

weight

spynnaker.pyNN.models.neural_properties package

Submodules

spynnaker.pyNN.models.neural_properties.neural_parameter module

class spynnaker.pyNN.models.neural_properties.neural_parameter.**NeuronParameter** (*value, data_type*)

Bases: object

get_dataspec_datatype ()

get_value ()

iterator_by_slice (*slice_start, slice_stop, spec*)
Creates an Iterator.

Parameters

- **slice_start** – Inclusive start of the range
- **slice_stop** – Exclusive end of the range
- **spec** (*DataSpecificationGenerator*) – The data specification to write to

Returns Iterator

Module contents

class spynnaker.pyNN.models.neural_properties.**NeuronParameter** (*value, data_type*)
Bases: object

get_dataspec_datatype ()

get_value ()

iterator_by_slice (*slice_start*, *slice_stop*, *spec*)
Creates an Iterator.

Parameters

- **slice_start** – Inclusive start of the range
- **slice_stop** – Exclusive end of the range
- **spec** (*DataSpecificationGenerator*) – The data specification to write to

Returns Iterator

spynnaker.pyNN.models.neuron package

Subpackages

spynnaker.pyNN.models.neuron.additional_inputs package

Submodules

spynnaker.pyNN.models.neuron.additional_inputs.abstract_additional_input module

class spynnaker.pyNN.models.neuron.additional_inputs.abstract_additional_input.**AbstractAddi**
Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.AbstractStandardNeuronComponent*

Represents a possible additional independent input for a model.

Parameters **data_types** – A list of data types in the component structure, in the order that they appear

spynnaker.pyNN.models.neuron.additional_inputs.additional_input_ca2_adaptive module

class spynnaker.pyNN.models.neuron.additional_inputs.additional_input_ca2_adaptive.**Additio**

Bases: *spynnaker.pyNN.models.neuron.additional_inputs.abstract_additional_input.AbstractAdditionalInput*

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

i_alpha

i_ca2

tau_ca2

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

Module contents

class spynnaker.pyNN.models.neuron.additional_inputs.**AbstractAdditionalInput** (*data_types*)
Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.AbstractStandardNeuronComponent*

Represents a possible additional independent input for a model.

Parameters **data_types** – A list of data types in the component structure, in the order that they appear

class spynnaker.pyNN.models.neuron.additional_inputs.**AdditionalInputCa2Adaptive** (*tau_ca2*, *i_ca2*, *i_alpha*)
Bases: *spynnaker.pyNN.models.neuron.additional_inputs.abstract_additional_input.AbstractAdditionalInput*

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (*str*) – The name of the variable

Return type bool

i_alpha

i_ca2

tau_ca2

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

spynnaker.pyNN.models.neuron.builds package

Submodules

spynnaker.pyNN.models.neuron.builds.eif_cond_alpha_isfa_ista module

class spynnaker.pyNN.models.neuron.builds.eif_cond_alpha_isfa_ista.**EIFConductanceAlphaPopul**

Bases: object

Exponential integrate and fire neuron with spike triggered and sub-threshold adaptation currents (isfa, ista reps.)

default_initial_values = {'gsyn_exc': 0.0, 'gsyn_inh': 0.0, 'v': -70.6, 'w': 0.0}

default_parameters = {'a': 4.0, 'b': 0.0805, 'cm': 0.281, 'delta_T': 2.0, 'e_rev_E'

spynnaker.pyNN.models.neuron.builds.hh_cond_exp module

class spynnaker.pyNN.models.neuron.builds.hh_cond_exp.**HHCondExp**(***kwargs*)

Bases: object

Single-compartment Hodgkin-Huxley model with exponentially decaying current input.

default_initial_values = {'gsyn_exc': 0.0, 'gsyn_inh': 0.0, 'v': -65.0}

default_parameters = {'cm': 0.2, 'e_rev_E': 0.0, 'e_rev_I': -80, 'e_rev_K': -90.0, 'e

spynnaker.pyNN.models.neuron.builds.if_cond_alpha module

class spynnaker.pyNN.models.neuron.builds.if_cond_alpha.**IFCondAlpha**(***kwargs*)

Bases: object

Leaky integrate and fire neuron with an alpha-shaped current input.

default_initial_values = {'gsyn_exc': 0.0, 'gsyn_inh': 0.0, 'v': -65.0}

default_parameters = {'cm': 1.0, 'e_rev_E': 0.0, 'e_rev_I': -70.0, 'i_offset': 0, 't

spynnaker.pyNN.models.neuron.builds.if_cond_exp_base module

class spynnaker.pyNN.models.neuron.builds.if_cond_exp_base.**IFCondExpBase**(***kwargs*)

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with an exponentially decaying conductance input.

spynnaker.pyNN.models.neuron.builds.if_cond_exp_stoc module

class spynnaker.pyNN.models.neuron.builds.if_cond_exp_stoc.**IFCondExpStoc**(***kwargs*)

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with a stochastic threshold.

spynnaker.pyNN.models.neuron.builds.if_curr_alpha module

class spynnaker.pyNN.models.neuron.builds.if_curr_alpha.**IFCurrAlpha**(***kwargs*)

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with an alpha-shaped current-based input.

spynnaker.pyNN.models.neuron.builds.if_curr_delta module

class spynnaker.pyNN.models.neuron.builds.if_curr_delta.**IFCurrDelta**(***kwargs*)

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with an instantaneous current input

spynnaker.pyNN.models.neuron.builds.if_curr_dual_exp_base module

class spynnaker.pyNN.models.neuron.builds.if_curr_dual_exp_base.**IFCurrDualExpBase**(***kwargs*)

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with two exponentially decaying excitatory current inputs, and one exponentially decaying inhibitory current input

spynnaker.pyNN.models.neuron.builds.if_curr_exp_base module

class spynnaker.pyNN.models.neuron.builds.if_curr_exp_base.**IFCurrExpBase**(***kwargs*)

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with an exponentially decaying current input

spynnaker.pyNN.models.neuron.builds.if_curr_exp_ca2_adaptive module

class spynnaker.pyNN.models.neuron.builds.if_curr_exp_ca2_adaptive.**IFCurrExpCa2Adaptive**(***k*

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Model from Liu, Y. H., & Wang, X. J. (2001). Spike-frequency adaptation of a generalized leaky integrate-and-fire model neuron. Journal of Computational Neuroscience, 10(1), 25-45. doi:10.1023/A:1008916026143

spynnaker.pyNN.models.neuron.builds.if_curr_exp_semd_base module

class spynnaker.pyNN.models.neuron.builds.if_curr_exp_semd_base.**IFCurrExpSEMDBase**(***kwargs*)

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with an exponentially decaying current input, where the excitatory input depends upon the inhibitory input (see https://www.cit-ec.de/en/nbs/spiking-insect-vision)

spynnaker.pyNN.models.neuron.builds.if_facets_hardware1 module

class spynnaker.pyNN.models.neuron.builds.if_facets_hardware1.**IFFacetsConductancePopulatio**

Bases: object

Leaky integrate and fire neuron with conductance-based synapses and fixed threshold as it is resembled by the FACETS Hardware Stage 1

default_initial_values = {'v': -65.0}

default_parameters = {'e_rev_I': -80, 'g_leak': 40.0, 'tau_syn_E': 30.0, 'tau_syn_I':

spynnaker.pyNN.models.neuron.builds.izk_cond_exp_base module

class spynnaker.pyNN.models.neuron.builds.izk_cond_exp_base.**IzkCondExpBase**(***kwargs*)

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

spynnaker.pyNN.models.neuron.builds.izk_curr_exp_base module

class spynnaker.pyNN.models.neuron.builds.izk_curr_exp_base.**IzkCurrExpBase**(***kwargs*)

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Module contents

class spynnaker.pyNN.models.neuron.builds.**EIFConductanceAlphaPopulation**(***kwargs*)

Bases: object

Exponential integrate and fire neuron with spike triggered and sub-threshold adaptation currents (isfa, ista reps.)

default_initial_values = {'gsyn_exc': 0.0, 'gsyn_inh': 0.0, 'v': -70.6, 'w': 0.0}

default_parameters = {'a': 4.0, 'b': 0.0805, 'cm': 0.281, 'delta_T': 2.0, 'e_rev_E'

class spynnaker.pyNN.models.neuron.builds.**HHCondExp**(***kwargs*)

Bases: object

Single-compartment Hodgkin-Huxley model with exponentially decaying current input.

default_initial_values = {'gsyn_exc': 0.0, 'gsyn_inh': 0.0, 'v': -65.0}

default_parameters = {'cm': 0.2, 'e_rev_E': 0.0, 'e_rev_I': -80, 'e_rev_K': -90.0, 'e

class spynnaker.pyNN.models.neuron.builds.**IFCondAlpha**(***kwargs*)

Bases: object

Leaky integrate and fire neuron with an alpha-shaped current input.

default_initial_values = {'gsyn_exc': 0.0, 'gsyn_inh': 0.0, 'v': -65.0}

default_parameters = {'cm': 1.0, 'e_rev_E': 0.0, 'e_rev_I': -70.0, 'i_offset': 0, 't

class spynnaker.pyNN.models.neuron.builds.**IFCondExpBase**(***kwargs*)

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with an exponentially decaying conductance input.

class spynnaker.pyNN.models.neuron.builds.**IFCurrAlpha**(***kwargs*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with an alpha-shaped current-based input.

class spynnaker.pyNN.models.neuron.builds.**IFCurrDualExpBase**(***kwargs*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with two exponentially decaying excitatory current inputs, and one exponentially decaying inhibitory current input

class spynnaker.pyNN.models.neuron.builds.**IFCurrExpBase**(***kwargs*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with an exponentially decaying current input

class spynnaker.pyNN.models.neuron.builds.**IFFacetsConductancePopulation**(***kwargs*)
Bases: object

Leaky integrate and fire neuron with conductance-based synapses and fixed threshold as it is resembled by the FACETS Hardware Stage 1

default_initial_values = {'v': -65.0}

default_parameters = {'e_rev_I': -80, 'g_leak': 40.0, 'tau_syn_E': 30.0, 'tau_syn_I':

class spynnaker.pyNN.models.neuron.builds.**IzkCondExpBase**(***kwargs*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

class spynnaker.pyNN.models.neuron.builds.**IzkCurrExpBase**(***kwargs*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

class spynnaker.pyNN.models.neuron.builds.**IFCondExpStoc**(***kwargs*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with a stochastic threshold.

class spynnaker.pyNN.models.neuron.builds.**IFCurrDelta**(***kwargs*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with an instantaneous current input

class spynnaker.pyNN.models.neuron.builds.**IFCurrExpCa2Adaptive**(***kwargs*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Model from Liu, Y. H., & Wang, X. J. (2001). Spike-frequency adaptation of a generalized leaky integrate-and-fire model neuron. Journal of Computational Neuroscience, 10(1), 25-45. doi:10.1023/A:1008916026143

class spynnaker.pyNN.models.neuron.builds.**IFCurrExpSEMDBase**(***kwargs*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard*

Leaky integrate and fire neuron with an exponentially decaying current input, where the excitatory input depends upon the inhibitory input (see https://www.cit-ec.de/en/nbs/spiking-insect-vision)

spynnaker.pyNN.models.neuron.implementations package

Submodules

spynnaker.pyNN.models.neuron.implementations.abstract_neuron_impl module

class spynnaker.pyNN.models.neuron.implementations.abstract_neuron_impl.**AbstractNeuronImpl**
Bases: object

An abstraction of a whole neuron model including all parts

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters state_variables (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

binary_name
The name of the binary executable of this implementation

:rtype str

get_data(*parameters*, *state_variables*, *vertex_slice*)
Get the data to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of the vertex to generate parameters for

Return type numpy array of uint32

get_dtcm_usage_in_bytes(*n_neurons*)
Get the DTCM memory usage required

Parameters n_neurons (*int*) – The number of neurons to get the usage for

Return type int

get_global_weight_scale()
Get the weight scaling required by this model

Return type int

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters n_neurons (*int*) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported by the model

Return type int

get_recordable_units(*variable*)
Get the units of the given variable that can be recorded

Parameters **variable** (*str*) – The name of the variable

get_recordable_variable_index(*variable*)
Get the index of the variable in the list of variables that can be recorded

Parameters **variable** (*str*) – The name of the variable

Return type int

get_recordable_variables()
Get the names of the variables that can be recorded in this model

Return type list of str

get_sdram_usage_in_bytes(*n_neurons*)
Get the SDRAM memory usage required

Parameters **n_neurons** (*int*) – The number of neurons to get the usage for

Return type int

get_synapse_id_by_target(*target*)
Get the id of a synapse given the name

Parameters **target** (*str*) – The name of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type

Return type array of str

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

is_conductance_based
Determine if the model uses conductance

Return type bool

is_recordable(*variable*)
Determine if the given variable can be recorded

Parameters **variable** (*str*) – The name of the variable being requested

Return type bool

model_name
The name of the model

Return type str

read_data(*data*, *offset*, *vertex_slice*, *parameters*, *state_variables*)
Read the parameters and state variables of the model from the given data

Parameters

- **data** – The data to be read
- **offset** – The offset where the data should be read from

- **vertex_slice** – The slice of the vertex to read parameters for
- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters to update
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables to update

spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component module

class spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.**Abst**

Bases: object

Represents a component of a standard neural model

Parameters data_types – A list of data types in the component structure, in the order that they appear

add_parameters (*parameters*)

Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)

Add the initial values of the state variables to the state variables holder

Parameters state_variables (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_data (*parameters*, *state_variables*, *vertex_slice*)

Get the data to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of the vertex to generate parameters for

Return type numpy array of uint32

get_dtcm_usage_in_bytes (*n_neurons*)

Get the DTCM memory usage required

Parameters n_neurons (*int*) – The number of neurons to get the usage for

Return type int

get_n_cpu_cycles (*n_neurons*)

Get the number of CPU cycles required to update the state

Parameters n_neurons (*int*) – The number of neurons to get the cycles for

Return type int

get_sdram_usage_in_bytes (*n_neurons*)

Get the SDRAM memory usage required

Parameters n_neurons (*int*) – The number of neurons to get the usage for

Return type int

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

read_data(*data*, *offset*, *vertex_slice*, *parameters*, *state_variables*)
Read the parameters and state variables of the model from the given data

Parameters

- **data** – The data to be read
- **offset** – The offset where the data should be read from
- **vertex_slice** – The slice of the vertex to read parameters for
- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters to update
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables to update

Returns The offset after reading the data

struct
The structure of the component

Return type :py:class:'spynnaker.pyNN.models.neuron.implementations.struct.Struct'

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

spynnaker.pyNN.models.neuron.implementations.neuron_impl_standard module

class spynnaker.pyNN.models.neuron.implementations.neuron_impl_standard.**NeuronImplStandard**

Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_neuron_impl.AbstractNeuronImpl*

The standard neuron implementation, consisting of various components

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

binary_name
The name of the binary executable of this implementation

:rtype str

get_data (*parameters*, *state_variables*, *vertex_slice*)
Get the data to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of the vertex to generate parameters for

Return type numpy array of uint32

get_dtcm_usage_in_bytes (*n_neurons*)
Get the DTCM memory usage required

Parameters **n_neurons** (*int*) – The number of neurons to get the usage for

Return type int

get_global_weight_scale ()
Get the weight scaling required by this model

Return type int

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported by the model

Return type int

get_recordable_units(*variable*)
Get the units of the given variable that can be recorded

Parameters **variable** (*str*) – The name of the variable

get_recordable_variable_index(*variable*)
Get the index of the variable in the list of variables that can be recorded

Parameters **variable** (*str*) – The name of the variable

Return type int

get_recordable_variables()
Get the names of the variables that can be recorded in this model

Return type list of str

get_sdram_usage_in_bytes(*n_neurons*)
Get the SDRAM memory usage required

Parameters **n_neurons** (*int*) – The number of neurons to get the usage for

Return type int

get_synapse_id_by_target(*target*)
Get the id of a synapse given the name

Parameters **target** (*str*) – The name of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type

Return type array of str

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

is_conductance_based
Determine if the model uses conductance

Return type bool

is_recordable(*variable*)
Determine if the given variable can be recorded

Parameters **variable** (*str*) – The name of the variable being requested

Return type bool

model_name
The name of the model

Return type str

read_data (*data*, *offset*, *vertex_slice*, *parameters*, *state_variables*)
Read the parameters and state variables of the model from the given data

Parameters

- **data** – The data to be read
- **offset** – The offset where the data should be read from
- **vertex_slice** – The slice of the vertex to read parameters for
- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters to update
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables to update

spynnaker.pyNN.models.neuron.implementations.ranged_dict_vertex_slice module

class `spynnaker.pyNN.models.neuron.implementations.ranged_dict_vertex_slice.`**RangedDictVerte**

Bases: `object`

A slice of a ranged dict to be used to update values

spynnaker.pyNN.models.neuron.implementations.struct module

class `spynnaker.pyNN.models.neuron.implementations.struct.`**Struct** (*field_types*)
Bases: `object`

Represents a C code structure

Parameters **field_types** (list of `data_specification.enums.data_type.DataType`) – The types of the fields, ordered as they appear in the struct

field_types
The types of the fields, ordered as they appear in the struct

Return type list of `data_specification.enums.data_type.DataType`

get_data (*values*, *offset=0*, *array_size=1*)
Get a numpy array of uint32 of data for the given values

Parameters

- **values** (*list of (single value or list of values or RangedList of values)*) – A list of values with length the same size as the number of fields returned by field_types
- **offset** – The offset into each of the values where to start
- **array_size** – The number of structs to generate

Return type numpy.array(dtype="uint32")

get_size_in_whole_words (*array_size=1*)
Get the size of the struct in whole words in an array of given size (default 1 item)

Parameters **array_size** – The number of elements in an array of structs

Return type int

numpy_dtype
The numpy data type of the struct

Return type numpy.dtype

read_data (*data*, *offset=0*, *array_size=1*)
Read a bytearray of data and convert to struct values

Parameters

- **data** – The data to be read
- **offset** – Index of the byte at the start of the valid data
- **array_size** – The number of struct elements to read

Returns a list of lists of data values, one list for each struct element

Module contents

class spynnaker.pyNN.models.neuron.implementations.**AbstractStandardNeuronComponent** (*data_types*
Bases: object

Represents a component of a standard neural model

Parameters **data_types** – A list of data types in the component structure, in the order that they appear

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_data (*parameters*, *state_variables*, *vertex_slice*)
Get the data to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of the vertex to generate parameters for

Return type numpy array of uint32

get_dtcm_usage_in_bytes (*n_neurons*)
Get the DTCM memory usage required

Parameters **n_neurons** (*int*) – The number of neurons to get the usage for

Return type int

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (`int`) – The number of neurons to get the cycles for

Return type int

get_sdram_usage_in_bytes (*n_neurons*)
Get the SDRAM memory usage required

Parameters **n_neurons** (`int`) – The number of neurons to get the usage for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

read_data (*data*, *offset*, *vertex_slice*, *parameters*, *state_variables*)
Read the parameters and state variables of the model from the given data

Parameters

- **data** – The data to be read
- **offset** – The offset where the data should be read from
- **vertex_slice** – The slice of the vertex to read parameters for
- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters to update
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables to update

Returns The offset after reading the data

struct
The structure of the component

Return type :py:class:'spynnaker.pyNN.models.neuron.implementations.struct.Struct'

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

class spynnaker.pyNN.models.neuron.implementations.**Struct** (*field_types*)
Bases: object

Represents a C code structure

Parameters **field_types** (list of data_specification.enums.data_type.DataType) – The types of the fields, ordered as they appear in the struct

field_types
The types of the fields, ordered as they appear in the struct

Return type list of data_specification.enums.data_type.DataType

get_data (*values*, *offset=0*, *array_size=1*)
Get a numpy array of uint32 of data for the given values

Parameters

- **values** (*list of (single value or list of values or RangedList of values)*) – A list of values with length the same size as the number of fields returned by field_types
- **offset** – The offset into each of the values where to start
- **array_size** – The number of structs to generate

Return type numpy.array(dtype="uint32")

get_size_in_whole_words (*array_size=1*)
Get the size of the struct in whole words in an array of given size (default 1 item)

Parameters **array_size** – The number of elements in an array of structs

Return type int

numpy_dtype
The numpy data type of the struct

Return type numpy.dtype

read_data (*data*, *offset=0*, *array_size=1*)
Read a bytearray of data and convert to struct values

Parameters

- **data** – The data to be read
- **offset** – Index of the byte at the start of the valid data
- **array_size** – The number of struct elements to read

Returns a list of lists of data values, one list for each struct element

class spynnaker.pyNN.models.neuron.implementations.**NeuronImplStandard**(*model_name*, *binary*, *neuron_model*, *input_type*, *synapse_type*, *threshold_type*, *additional_input_type=None*)

Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_neuron_impl.AbstractNeuronImpl*

The standard neuron implementation, consisting of various components

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

binary_name
The name of the binary executable of this implementation

:rtype str

get_data(*parameters*, *state_variables*, *vertex_slice*)
Get the data to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of the vertex to generate parameters for

Return type numpy array of uint32

get_dtcm_usage_in_bytes(*n_neurons*)
Get the DTCM memory usage required

Parameters **n_neurons** (*int*) – The number of neurons to get the usage for

Return type int

get_global_weight_scale()
Get the weight scaling required by this model

Return type int

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported by the model

Return type int

get_recordable_units(*variable*)
Get the units of the given variable that can be recorded

Parameters **variable** (*str*) – The name of the variable

get_recordable_variable_index(*variable*)
Get the index of the variable in the list of variables that can be recorded

Parameters **variable** (*str*) – The name of the variable

Return type int

get_recordable_variables()
Get the names of the variables that can be recorded in this model

Return type list of str

get_sdram_usage_in_bytes(*n_neurons*)
Get the SDRAM memory usage required

Parameters **n_neurons** (*int*) – The number of neurons to get the usage for

Return type int

get_synapse_id_by_target(*target*)
Get the id of a synapse given the name

Parameters **target** (*str*) – The name of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type

Return type array of str

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

is_conductance_based
Determine if the model uses conductance

Return type bool

is_recordable(*variable*)
Determine if the given variable can be recorded

Parameters **variable** (*str*) – The name of the variable being requested

Return type bool

model_name
The name of the model

Return type str

read_data (*data*, *offset*, *vertex_slice*, *parameters*, *state_variables*)
Read the parameters and state variables of the model from the given data

Parameters

- **data** – The data to be read
- **offset** – The offset where the data should be read from
- **vertex_slice** – The slice of the vertex to read parameters for
- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters to update
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables to update

class `spynnaker.pyNN.models.neuron.implementations.`**AbstractNeuronImpl**
Bases: `object`

An abstraction of a whole neuron model including all parts

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters state_variables (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the state variables

binary_name
The name of the binary executable of this implementation

:rtype str

get_data (*parameters*, *state_variables*, *vertex_slice*)
Get the data to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of the vertex to generate parameters for

Return type numpy array of uint32

get_dtcm_usage_in_bytes (*n_neurons*)
Get the DTCM memory usage required

Parameters n_neurons (*int*) – The number of neurons to get the usage for

Return type int

get_global_weight_scale ()
Get the weight scaling required by this model

Return type int

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported by the model

Return type int

get_recordable_units(*variable*)
Get the units of the given variable that can be recorded

Parameters **variable** (*str*) – The name of the variable

get_recordable_variable_index(*variable*)
Get the index of the variable in the list of variables that can be recorded

Parameters **variable** (*str*) – The name of the variable

Return type int

get_recordable_variables()
Get the names of the variables that can be recorded in this model

Return type list of str

get_sdram_usage_in_bytes(*n_neurons*)
Get the SDRAM memory usage required

Parameters **n_neurons** (*int*) – The number of neurons to get the usage for

Return type int

get_synapse_id_by_target(*target*)
Get the id of a synapse given the name

Parameters **target** (*str*) – The name of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type

Return type array of str

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

is_conductance_based
Determine if the model uses conductance

Return type bool

is_recordable(*variable*)
Determine if the given variable can be recorded

Parameters **variable** (*str*) – The name of the variable being requested

Return type bool

model_name
The name of the model

Return type str

read_data (*data*, *offset*, *vertex_slice*, *parameters*, *state_variables*)
Read the parameters and state variables of the model from the given data

Parameters

- **data** – The data to be read
- **offset** – The offset where the data should be read from
- **vertex_slice** – The slice of the vertex to read parameters for
- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters to update
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables to update

class spynnaker.pyNN.models.neuron.implementations.**RangedDictVertexSlice** (*ranged_dict*, *vertex_slice*)

Bases: object

A slice of a ranged dict to be used to update values

spynnaker.pyNN.models.neuron.input_types package

Submodules

spynnaker.pyNN.models.neuron.input_types.abstract_input_type module

class spynnaker.pyNN.models.neuron.input_types.abstract_input_type.**AbstractInputType** (*data_typ*
Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.AbstractStandardNeuronComponent*

Represents a possible input type for a neuron model (e.g., current).

Parameters data_types – A list of data types in the component structure, in the order that they appear

get_global_weight_scale ()
Get the global weight scaling value.

Returns The global weight scaling value

Return type float

spynnaker.pyNN.models.neuron.input_types.input_type_conductance module

class spynnaker.pyNN.models.neuron.input_types.input_type_conductance.**InputTypeConductance**

Bases: *spynnaker.pyNN.models.neuron.input_types.abstract_input_type.AbstractInputType*

The conductance input type

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

e_rev_E

e_rev_I

get_global_weight_scale()
Get the global weight scaling value.

Returns The global weight scaling value

Return type float

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (*str*) – The name of the variable

Return type bool

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

spynnaker.pyNN.models.neuron.input_types.input_type_current module

class spynnaker.pyNN.models.neuron.input_types.input_type_current.**InputTypeCurrent**

Bases: *spynnaker.pyNN.models.neuron.input_types.abstract_input_type.AbstractInputType*

The current input type

add_parameters(*parameters*)

Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)

Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_global_weight_scale()

Get the global weight scaling value.

Returns The global weight scaling value

Return type float

get_n_cpu_cycles(*n_neurons*)

Get the number of CPU cycles required to update the state

Parameters **n_neurons** (int) – The number of neurons to get the cycles for

Return type int

get_units(*variable*)

Get the units of the given variable

Parameters **variable** (str) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*)

Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)

Determine if this component has a variable by the given name

Parameters **variable** (str) – The name of the variable

Return type bool

update_values(*values*, *parameters*, *state_variables*)

Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

spynnaker.pyNN.models.neuron.input_types.input_type_current_semd module

class spynnaker.pyNN.models.neuron.input_types.input_type_current_semd.**InputTypeCurrentSEMD**

Bases: *spynnaker.pyNN.models.neuron.input_types.abstract_input_type.AbstractInputType*

The current sEMD input type

add_parameters (*parameters*)
: Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
: Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_global_weight_scale ()
: Get the global weight scaling value.

Returns The global weight scaling value

Return type float

get_n_cpu_cycles (*n_neurons*)
: Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
: Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*)
: Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

inh_input_previous

multiplicator

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

Module contents

class spynnaker.pyNN.models.neuron.input_types.**AbstractInputType**(*data_types*)
Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.AbstractStandardNeuronComponent*

Represents a possible input type for a neuron model (e.g., current).

Parameters **data_types** – A list of data types in the component structure, in the order that they appear

get_global_weight_scale()
Get the global weight scaling value.

Returns The global weight scaling value

Return type float

class spynnaker.pyNN.models.neuron.input_types.**InputTypeConductance**(*e_rev_E*, *e_rev_I*)
Bases: *spynnaker.pyNN.models.neuron.input_types.abstract_input_type.AbstractInputType*

The conductance input type

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

e_rev_E

e_rev_I

get_global_weight_scale()
Get the global weight scaling value.

Returns The global weight scaling value

Return type float

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (`int`) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

class `spynnaker.pyNN.models.neuron.input_types.`**InputTypeCurrent**
Bases: *spynnaker.pyNN.models.neuron.input_types.abstract_input_type.AbstractInputType*

The current input type

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the state variables

get_global_weight_scale()
Get the global weight scaling value.

Returns The global weight scaling value

Return type float

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (`int`) – The number of neurons to get the cycles for

Return type int

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

class spynnaker.pyNN.models.neuron.input_types.**InputTypeCurrentSEMD**(*multiplicator*, *inh_input_previous*)
Bases: *spynnaker.pyNN.models.neuron.input_types.abstract_input_type.AbstractInputType*

The current sEMD input type

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_global_weight_scale ()
Get the global weight scaling value.

Returns The global weight scaling value

Return type float

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (*str*) – The name of the variable

Return type bool

inh_input_previous

multiplicator

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

spynnaker.pyNN.models.neuron.master_pop_table_generators package

Submodules

spynnaker.pyNN.models.neuron.master_pop_table_generators.abstract_master_pop_table_factory module

class spynnaker.pyNN.models.neuron.master_pop_table_generators.abstract_master_pop_table_fa

Bases: object

extract_synaptic_matrix_data_location(*incoming_key*, *master_pop_base_mem_address*, *txrx*, *chip_x*, *chip_y*)

Parameters

- **incoming_key** (*int*) – the source key which the synaptic matrix needs to be mapped to
- **master_pop_base_mem_address** (*int*) – the base address of the master pop
- **txrx** (spinnman.transceiver.Transceiver) – the transceiver object
- **chip_y** (*int*) – the y coordinate of the chip of this master pop
- **chip_x** (*int*) – the x coordinate of the chip of this master pop

Returns a synaptic matrix memory position.

finish_master_pop_table(*spec*, *master_pop_table_region*)

Complete the master pop table in the data specification.

Parameters

- **spec** – the data specification to write the master pop entry to
- **master_pop_table_region** – the region to which the master pop table is being stored

get_edge_constraints()

Gets the constraints for this table on edges coming in to a vertex.

Returns a list of constraints

Return type list(pacman.model.constraints.AbstractConstraint)

get_master_population_table_size(*vertex_slice*, *in_edges*)

Get the size of the master population table in SDRAM

update_master_population_table(*spec*, *block_start_addr*, *row_length*, *key_and_mask*, *master_pop_table_region*, *is_single=False*)

Update a data specification with a master pop entry in some form

Parameters

- **spec** – the data specification to write the master pop entry to
- **block_start_addr** – the start address of the row in the region
- **row_length** – the row length of this entry
- **key_and_mask** (pacman.model.routing_info.BaseKeyAndMask) – a key_and_mask object used as part of describing an edge that will require being received to be stored in the master pop table; the whole edge will become multiple calls to this function

- **master_pop_table_region** – The region to which the master pop table is being stored
- **is_single** – True if this is a single synapse, False otherwise

spynnaker.pyNN.models.neuron.master_pop_table_generators.master_pop_table_as_binary_search module

class spynnaker.pyNN.models.neuron.master_pop_table_generators.master_pop_table_as_binary_s
Bases: *spynnaker.pyNN.models.neuron.master_pop_table_generators.abstract_master_pop_table_factory.AbstractMasterPopTableFactory*

Master population table, implemented as binary search master.

ADDRESS_LIST_DTYPE = '<u4'

ADDRESS_MASK = 2147483392

ADDRESS_SCALE = 16

ADDRESS_SCALED_SHIFT = 4

MASTER_POP_ENTRY_DTYPE = [('key', '<u4'), ('mask', '<u4'), ('start', '<u2'), ('count',

ROW_LENGTH_MASK = 255

SINGLE_BIT_FLAG_BIT = 2147483648

extract_synaptic_matrix_data_location(*incoming_key*, *master_pop_base_mem_address*, *txrx*, *chip_x*, *chip_y*)

Parameters

- **incoming_key** (*int*) – the source key which the synaptic matrix needs to be mapped to
- **master_pop_base_mem_address** (*int*) – the base address of the master pop
- **txrx** (spinnman.transceiver.Transceiver) – the transceiver object
- **chip_y** (*int*) – the y coordinate of the chip of this master pop
- **chip_x** (*int*) – the x coordinate of the chip of this master pop

Returns a synaptic matrix memory position.

finish_master_pop_table(*spec*, *master_pop_table_region*)
Complete the master pop table in the data specification.

Parameters

- **spec** – the data specification to write the master pop entry to
- **master_pop_table_region** – the region to which the master pop table is being stored

get_allowed_row_length(*row_length*)

Parameters **row_length** – the row length being considered

Returns the row length available

get_edge_constraints()
Gets the constraints for this table on edges coming in to a vertex.

Returns a list of constraints

Return type list(pacman.model.constraints.AbstractConstraint)

get_exact_master_population_table_size(*vertex*, *machine_graph*, *graph_mapper*)

Returns the size the master pop table will take in SDRAM (in bytes)

get_master_population_table_size(*vertex_slice*, *in_edges*)
Get the size of the master population table in SDRAM

Parameters

- **vertex_slice** – the slice of the vertex
- **in_edges** – the in coming edges

Returns the size the master pop table will take in SDRAM (in bytes)

get_next_allowed_address(*next_address*)

Parameters **next_address** – The next address that would be used

Returns The next address that can be used following next_address

initialise_table(*spec*, *master_population_table_region*)
Initialise the master pop data structure

Parameters

- **spec** – the DSG writer
- **master_population_table_region** – the region in memory that the master pop table will be written in

Return type None

update_master_population_table(*spec*, *block_start_addr*, *row_length*, *key_and_mask*, *master_pop_table_region*, *is_single=False*)
Add an entry in the binary search to deal with the synaptic matrix

Parameters

- **spec** – the writer for DSG
- **block_start_addr** – where the synaptic matrix block starts
- **row_length** – how long in bytes each synaptic entry is
- **key_and_mask** – the key and mask for this master pop entry
- **master_pop_table_region** – the region ID for the master pop
- **is_single** – Flag that states if the entry is a direct entry for a single row.

Returns The index of the entry, to be used to retrieve it

Return type int

Module contents

class spynnaker.pyNN.models.neuron.master_pop_table_generators.**MasterPopTableAsBinarySearcl**
Bases: *spynnaker.pyNN.models.neuron.master_pop_table_generators.abstract_master_pop_table_factory.AbstractMasterPopTableFactory*

Master population table, implemented as binary search master.

ADDRESS_LIST_DTYPE = '<u4'

ADDRESS_MASK = 2147483392

ADDRESS_SCALE = 16

ADDRESS_SCALED_SHIFT = 4

MASTER_POP_ENTRY_DTYPE = [('key', '<u4'), ('mask', '<u4'), ('start', '<u2'), ('count',

ROW_LENGTH_MASK = 255

SINGLE_BIT_FLAG_BIT = 2147483648

extract_synaptic_matrix_data_location(*incoming_key*, *master_pop_base_mem_address*, *txrx*, *chip_x*, *chip_y*)

Parameters

- **incoming_key** (`int`) – the source key which the synaptic matrix needs to be mapped to
- **master_pop_base_mem_address** (`int`) – the base address of the master pop
- **txrx** (`spinnman.transceiver.Transceiver`) – the transceiver object
- **chip_y** (`int`) – the y coordinate of the chip of this master pop
- **chip_x** (`int`) – the x coordinate of the chip of this master pop

Returns a synaptic matrix memory position.

finish_master_pop_table(*spec*, *master_pop_table_region*)

Complete the master pop table in the data specification.

Parameters

- **spec** – the data specification to write the master pop entry to
- **master_pop_table_region** – the region to which the master pop table is being stored

get_allowed_row_length(*row_length*)

Parameters **row_length** – the row length being considered

Returns the row length available

get_edge_constraints()

Gets the constraints for this table on edges coming in to a vertex.

Returns a list of constraints

Return type list(`pacman.model.constraints.AbstractConstraint`)

get_exact_master_population_table_size(*vertex*, *machine_graph*, *graph_mapper*)

Returns the size the master pop table will take in SDRAM (in bytes)

get_master_population_table_size(*vertex_slice*, *in_edges*)

Get the size of the master population table in SDRAM

Parameters

- **vertex_slice** – the slice of the vertex
- **in_edges** – the in coming edges

Returns the size the master pop table will take in SDRAM (in bytes)

get_next_allowed_address(*next_address*)

Parameters **next_address** – The next address that would be used

Returns The next address that can be used following next_address

initialise_table(*spec*, *master_population_table_region*)
Initialise the master pop data structure

Parameters

- **spec** – the DSG writer
- **master_population_table_region** – the region in memory that the master pop table will be written in

Return type None

update_master_population_table(*spec*, *block_start_addr*, *row_length*, *key_and_mask*, *master_pop_table_region*, *is_single=False*)
Add an entry in the binary search to deal with the synaptic matrix

Parameters

- **spec** – the writer for DSG
- **block_start_addr** – where the synaptic matrix block starts
- **row_length** – how long in bytes each synaptic entry is
- **key_and_mask** – the key and mask for this master pop entry
- **master_pop_table_region** – the region ID for the master pop
- **is_single** – Flag that states if the entry is a direct entry for a single row.

Returns The index of the entry, to be used to retrieve it

Return type int

spynnaker.pyNN.models.neuron.neuron_models package

Submodules

spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model module

class spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model.**AbstractNeuronModel**

Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.AbstractStandardNeuronComponent*

Represents a neuron model.

Parameters

- **data_types** – A list of data types in the neuron structure, in the order that they appear
- **global_data_types** – A list of data types in the neuron global structure, in the order that they appear

get_data(*parameters*, *state_variables*, *vertex_slice*)
Get the data to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of the vertex to generate parameters for

Return type numpy array of uint32

get_dtcm_usage_in_bytes(*n_neurons*)
Get the DTCM memory usage required

Parameters **n_neurons** (int) – The number of neurons to get the usage for

Return type int

get_global_values()
Get the global values to be written to the machine for this model

Returns A list with the same length as self.global_struct.field_types

Return type A list of single values

get_sdram_usage_in_bytes(*n_neurons*)
Get the SDRAM memory usage required

Parameters **n_neurons** (int) – The number of neurons to get the usage for

Return type int

global_struct
Get the global parameters structure

read_data(*data*, *offset*, *vertex_slice*, *parameters*, *state_variables*)
Read the parameters and state variables of the model from the given data

Parameters

- **data** – The data to be read
- **offset** – The offset where the data should be read from
- **vertex_slice** – The slice of the vertex to read parameters for
- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters to update
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables to update

Returns The offset after reading the data

spynnaker.pyNN.models.neuron.neuron_models.neuron_model_izh module

class spynnaker.pyNN.models.neuron.neuron_models.neuron_model_izh.**NeuronModelIzh**(*a*, *b*, *c*, *d*, *v_init*, *u_init*, *i_offset*)

Bases: *spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model.AbstractNeuronModel*

a

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters state_variables (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

b

c

d

get_global_values(*machine_time_step*)
Get the global values to be written to the machine for this model

Returns A list with the same length as self.global_struct.field_types

Return type A list of single values

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters n_neurons (*int*) – The number of neurons to get the cycles for

Return type int

get_units(*variable*)
Get the units of the given variable

Parameters variable (*str*) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters variable (*str*) – The name of the variable

Return type bool

i_offset

u_init

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

v_init

spynnaker.pyNN.models.neuron.neuron_models.neuron_model_leaky_integrate_and_fire module

class spynnaker.pyNN.models.neuron.neuron_models.neuron_model_leaky_integrate_and_fire.**Neur**

Bases: *spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model.AbstractNeuronModel*

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters state_variables (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

cm

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters n_neurons (*int*) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters variable (*str*) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)

Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

i_offset

tau_m

tau_refrac

update_values (*values*, *parameters*, *state_variables*)

Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

v_init

v_reset

v_rest

Module contents

class spynnaker.pyNN.models.neuron.neuron_models.**AbstractNeuronModel** (*data_types*, *global_data_types=None*)

Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.AbstractStandardNeuronComponent*

Represents a neuron model.

Parameters

- **data_types** – A list of data types in the neuron structure, in the order that they appear
- **global_data_types** – A list of data types in the neuron global structure, in the order that they appear

get_data (*parameters*, *state_variables*, *vertex_slice*)

Get the data to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of the vertex to generate parameters for

Return type numpy array of uint32

get_dtcm_usage_in_bytes(*n_neurons*)
Get the DTCM memory usage required

Parameters **n_neurons** (`int`) – The number of neurons to get the usage for

Return type int

get_global_values()
Get the global values to be written to the machine for this model

Returns A list with the same length as self.global_struct.field_types

Return type A list of single values

get_sdram_usage_in_bytes(*n_neurons*)
Get the SDRAM memory usage required

Parameters **n_neurons** (`int`) – The number of neurons to get the usage for

Return type int

global_struct
Get the global parameters structure

read_data(*data*, *offset*, *vertex_slice*, *parameters*, *state_variables*)
Read the parameters and state variables of the model from the given data

Parameters

- **data** – The data to be read
- **offset** – The offset where the data should be read from
- **vertex_slice** – The slice of the vertex to read parameters for
- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters to update
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables to update

Returns The offset after reading the data

class spynnaker.pyNN.models.neuron.neuron_models.**NeuronModelIzh**(*a*, *b*, *c*, *d*, *v_init*, *u_init*, *i_offset*)
Bases: *spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model.AbstractNeuronModel*

a

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the state variables

b

c

d

get_global_values (*machine_time_step*)
Get the global values to be written to the machine for this model

Returns A list with the same length as self.global_struct.field_types

Return type A list of single values

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (`int`) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

i_offset

u_init

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

v_init

class spynnaker.pyNN.models.neuron.neuron_models.**NeuronModelLeakyIntegrateAndFire**(*v_init*, *v_rest*, *tau_m*, *cm*, *i_offset*, *v_reset*, *tau_refrac*)

Bases: *spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model.AbstractNeuronModel*

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

cm

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (*str*) – The name of the variable

Return type bool

i_offset

tau_m

tau_refrac

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

v_init

v_reset

v_rest

spynnaker.pyNN.models.neuron.plasticity package

Subpackages

spynnaker.pyNN.models.neuron.plasticity.stdp package

Subpackages

spynnaker.pyNN.models.neuron.plasticity.stdp.common package

Submodules

spynnaker.pyNN.models.neuron.plasticity.stdp.common.plasticity_helpers module

spynnaker.pyNN.models.neuron.plasticity.stdp.common.plasticity_helpers.**float_to_fixed** (*value*, *fixed_p*

spynnaker.pyNN.models.neuron.plasticity.stdp.common.plasticity_helpers.**get_lut_provenance** (*p* *p* *r* *e* *t* *p* *l*

spynnaker.pyNN.models.neuron.plasticity.stdp.common.plasticity_helpers.**write_exp_lut** (*spec*, *time_co* *size*, *shift*, *fixed_po*

Module contents

spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure package

Submodules

spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.abstract_synapse_structure module

class spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.abstract_synapse_struc

Bases: object

get_n_half_words_per_connection()

Get the number of bytes for each connection

get_weight_half_word()

The index of the half-word where the weight should be written

spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.synapse_structure_weight_accumulator module

class spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.synapse_structure_weig

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.abstract_synapse_structure.AbstractSynapseStructure*

get_n_half_words_per_connection()

Get the number of bytes for each connection

get_weight_half_word()

The index of the half-word where the weight should be written

spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.synapse_structure_weight_only module

class spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.synapse_structure_weig

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.abstract_synapse_structure.AbstractSynapseStructure*

get_n_half_words_per_connection()

Get the number of bytes for each connection

get_weight_half_word()

The index of the half-word where the weight should be written

Module contents

class spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.**AbstractSynapseStructu**

Bases: object

get_n_half_words_per_connection()

Get the number of bytes for each connection

get_weight_half_word()

The index of the half-word where the weight should be written

class spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.**SynapseStructureWeight**

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.abstract_synapse_structure.AbstractSynapseStructure*

get_n_half_words_per_connection()

Get the number of bytes for each connection

get_weight_half_word()
The index of the half-word where the weight should be written

class spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.**SynapseStructureWeight**
Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure.abstract_synapse_structure.AbstractSynapseStructure*

get_n_half_words_per_connection()
Get the number of bytes for each connection

get_weight_half_word()
The index of the half-word where the weight should be written

spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence package

Submodules

spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence module

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_depend
Bases: object

get_parameter_names()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

get_provenance_data(*pre_population_label*, *post_population_label*)
Get any provenance data

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters(*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_pfister_spike_triplet module

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_pfis

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence*

get_parameter_names()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

get_provenance_data(*pre_population_label*, *post_population_label*)
Get any provenance data

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

tau_minus

tau_plus

tau_x

tau_y

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters(*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_recurrent module

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_recu

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence*

default_parameters = {'accumulator_depression': -6, 'accumulator_potentiation': 6, '

get_parameter_names()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters(*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spike_nearest_pair module

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spik

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence*

default_parameters = {'tau_minus': 20.0, 'tau_plus': 20.0}

get_parameter_names()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

get_provenance_data(*pre_population_label*, *post_population_label*)
Get any provenance data

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

tau_minus

tau_plus

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters(*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spike_pair module

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spil

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence*

get_parameter_names()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

get_provenance_data(*pre_population_label*, *post_population_label*)
Get any provenance data

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

tau_minus

tau_plus

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters (*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_vogels_2011 module

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_voge

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence*

alpha

default_parameters = {'tau': 20.0}

get_parameter_names ()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes ()
Get the amount of SDRAM used by the parameters of this rule

is_same_as (*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

tau

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters (*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

Module contents

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**AbstractTimingDepender**
Bases: object

get_parameter_names ()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes ()
Get the amount of SDRAM used by the parameters of this rule

get_provenance_data (*pre_population_label*, *post_population_label*)
Get any provenance data

is_same_as (*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters (*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**TimingDependenceSpikeI**

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence*

get_parameter_names ()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes ()
Get the amount of SDRAM used by the parameters of this rule

get_provenance_data (*pre_population_label*, *post_population_label*)
Get any provenance data

is_same_as (*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

tau_minus

tau_plus

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters (*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**TimingDependencePfiste**

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence*

get_parameter_names()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

get_provenance_data(*pre_population_label*, *post_population_label*)
Get any provenance data

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

tau_minus

tau_plus

tau_x

tau_y

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters(*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**TimingDependenceRecur**

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence*

default_parameters = {'accumulator_depression': -6, 'accumulator_potentiation': 6, '

get_parameter_names()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters (*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**TimingDependenceSpikeI**

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence*

default_parameters = {'tau_minus': 20.0, 'tau_plus': 20.0}

get_parameter_names ()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes ()
Get the amount of SDRAM used by the parameters of this rule

get_provenance_data (*pre_population_label*, *post_population_label*)
Get any provenance data

is_same_as (*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

tau_minus

tau_plus

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters (*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**TimingDependenceVogels**

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence*

alpha

default_parameters = {'tau': 20.0}

get_parameter_names()
Return the names of the parameters supported by this timing dependency model.

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

n_weight_terms
The number of weight terms expected by this timing rule

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

synaptic_structure
Get the synaptic structure of the plastic part of the rows

tau

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters(*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence package

Submodules

spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_has_a_plus_a_minus module

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_has_a_plus_a_
Bases: object

A_minus

A_plus

set_a_plus_a_minus(*a_plus*, *a_minus*)

spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_weight_dependence module

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_weight_depend
Bases: object

get_parameter_names()
Returns the parameter names

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_synapse_types*, *n_weight_terms*)
Get the amount of SDRAM used by the parameters of this rule

get_provenance_data (*pre_population_label*, *post_population_label*)
Get any provenance data

Parameters

- **pre_population_label** – label of pre.
- **post_population_label** – label of post.

Returns the provenance data of the weight dependency

is_same_as (*weight_dependence*)
Determine if this weight dependence is the same as another

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

write_parameters (*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_additive module

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_add

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_has_a_plus_a_minus.AbstractHasAPlusAMinus*, *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_weight_dependence.AbstractWeightDependence*

get_parameter_names ()
Returns the parameter names

Return type iterable(str)

get_parameters_sdram_usage_in_bytes (*n_synapse_types*, *n_weight_terms*)
Get the amount of SDRAM used by the parameters of this rule

is_same_as (*weight_dependence*)
Determine if this weight dependence is the same as another

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

w_max

w_min

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

write_parameters (*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_additive_triplet module

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_add

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_has_a_plus_a_minus.AbstractHasAPlusAMinus*, *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_weight_dependence.AbstractWeightDependence*

A3_minus

A3_plus

default_parameters = {'A3_minus': 0.01, 'A3_plus': 0.01, 'w_max': 1.0, 'w_min': 0.

get_parameter_names()

Returns the parameter names

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_synapse_types*, *n_weight_terms*)

Get the amount of SDRAM used by the parameters of this rule

is_same_as(*weight_dependence*)

Determine if this weight dependence is the same as another

vertex_executable_suffix

The suffix to be appended to the vertex executable for this rule

w_max

w_min

weight_maximum

The maximum weight that will ever be set in a synapse as a result of this rule

write_parameters(*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)

Write the parameters of the rule to the spec

spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_multiplicative module

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_mult

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_has_a_plus_a_minus.AbstractHasAPlusAMinus*, *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_weight_dependence.AbstractWeightDependence*

get_parameter_names()

Returns the parameter names

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_synapse_types*, *n_weight_terms*)

Get the amount of SDRAM used by the parameters of this rule

is_same_as (*weight_dependence*)
Determine if this weight dependence is the same as another

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

w_max

w_min

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

write_parameters (*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

Module contents

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.**AbstractHasAPlusAMinus**
Bases: object

A_minus

A_plus

set_a_plus_a_minus (*a_plus*, *a_minus*)

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.**AbstractWeightDependence**
Bases: object

get_parameter_names ()
Returns the parameter names

Return type iterable(str)

get_parameters_sdram_usage_in_bytes (*n_synapse_types*, *n_weight_terms*)
Get the amount of SDRAM used by the parameters of this rule

get_provenance_data (*pre_population_label*, *post_population_label*)
Get any provenance data

Parameters

- **pre_population_label** – label of pre.
- **post_population_label** – label of post.

Returns the provenance data of the weight dependency

is_same_as (*weight_dependence*)
Determine if this weight dependence is the same as another

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

write_parameters (*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.**WeightDependenceAddit**

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_has_a_plus_a_minus.AbstractHasAPlusAMinus*, *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_weight_dependence.AbstractWeightDependence*

get_parameter_names()
Returns the parameter names

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_synapse_types*, *n_weight_terms*)
Get the amount of SDRAM used by the parameters of this rule

is_same_as(*weight_dependence*)
Determine if this weight dependence is the same as another

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

w_max

w_min

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

write_parameters(*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.**WeightDependenceMultip**

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_has_a_plus_a_minus.AbstractHasAPlusAMinus*, *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_weight_dependence.AbstractWeightDependence*

get_parameter_names()
Returns the parameter names

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_synapse_types*, *n_weight_terms*)
Get the amount of SDRAM used by the parameters of this rule

is_same_as(*weight_dependence*)
Determine if this weight dependence is the same as another

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

w_max

w_min

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

write_parameters(*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.**WeightDependenceAddit**

Bases: *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_has_a_plus_a_minus.AbstractHasAPlusAMinus*, *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.abstract_weight_dependence.AbstractWeightDependence*

A3_minus

A3_plus

default_parameters = {'A3_minus': 0.01, 'A3_plus': 0.01, 'w_max': 1.0, 'w_min': 0.

get_parameter_names()
Returns the parameter names

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_synapse_types*, *n_weight_terms*)
Get the amount of SDRAM used by the parameters of this rule

is_same_as(*weight_dependence*)
Determine if this weight dependence is the same as another

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

w_max

w_min

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

write_parameters(*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

Module contents

Module contents

spynnaker.pyNN.models.neuron.synapse_dynamics package

Submodules

spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_generate_on_machine module

class spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_generate_on_machine.**AbstractGe**
Bases: object

A synapse dynamics that can be generated on the machine

gen_matrix_id
The ID of the on-machine matrix generator

Return type int

gen_matrix_params
Any parameters required by the matrix generator

Return type numpy array of uint32

gen_matrix_params_size_in_bytes
The size of the parameters of the matrix generator in bytes

Return type int

generate_on_machine()
Determines if this instance should be generated on the machine.

Default implementation returns True

Return type bool

class spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_generate_on_machine.**MatrixGene**
Bases: enum.Enum

An enumeration.

STATIC_MATRIX = 0

STDP_MATRIX = 1

spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_plastic_synapse_dynamics module

class spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_plastic_synapse_dynamics.**Abst**
Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics.AbstractSynapseDynamics*

Synapses which change over time

get_n_fixed_plastic_words_per_row(*fp_size*)
Get the number of fixed plastic words to be read from each row

get_n_plastic_plastic_words_per_row(*pp_size*)
Get the number of plastic plastic words to be read from each row

get_n_synapses_in_rows(*pp_size*, *fp_size*)
Get the number of synapses in each of the rows with plastic sizes pp_size and fp_size

get_n_words_for_plastic_connections(*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_plastic_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*)
Get the fixed-plastic data, and plastic-plastic data for each row, and lengths for the fixed_plastic and plastic-plastic parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row, for each of the fixed-plastic and plastic-plastic data regions. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for each of the fixed-plastic and plastic-plastic regions.

read_plastic_synaptic_data(*post_vertex_slice*, *n_synapse_types*, *pp_size*, *pp_data*, *fp_size*, *fp_data*)
Read the connections indicated in the connection indices from the data in pp_data and fp_data

spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_static_synapse_dynamics module

class spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_static_synapse_dynamics.**Abstra**

Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics.AbstractSynapseDynamics*

Dynamics which don't change over time.

get_n_static_words_per_row (*ff_size*)
Get the number of bytes to be read per row for the static data given the size that was written to each row

get_n_synapses_in_rows (*ff_size*)
Get the number of synapses in the rows with sizes ff_size

get_n_words_for_static_connections (*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_static_synaptic_data (*connections, connection_row_indices, n_rows, post_vertex_slice, n_synapse_types*)
Get the fixed-fixed data for each row, and lengths for the fixed-fixed parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row for the fixed-fixed region. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for the fixed-fixed region.

read_static_synaptic_data (*post_vertex_slice, n_synapse_types, ff_size, ff_data*)
Read the connections from the words of data in ff_data

spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics module

class spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics.**AbstractSynap**

Bases: object

NUMPY_CONNECTORS_DTYPE = [('source', 'uint32'), ('target', 'uint32'), ('weight', 'floa

are_weights_signed ()
Determines if the weights are signed values

changes_during_run
Determine if the synapses change during a run

Return type bool

convert_per_connection_data_to_rows (*connection_row_indices, n_rows, data*)
Converts per-connection data generated from connections into row-based data to be returned from get_synaptic_data

get_delay_maximum (*connector, delays*)
Get the maximum delay for the synapses

get_delay_variance (*connector, delays*)
Get the variance in delay for the synapses

get_max_synapses (*n_words*)
Get the maximum number of synapses that can be held in the given number of words

Parameters **n_words** – The number of words the synapses must fit in

Return type int

get_n_items (*rows*, *item_size*)
Get the number of items in each row as 4-byte values, given the item size

get_parameter_names ()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_parameters_sdram_usage_in_bytes (*n_neurons*, *n_synapse_types*)
Get the SDRAM usage of the synapse dynamics parameters in bytes

get_provenance_data (*pre_population_label*, *post_population_label*)
Get the provenance data from this synapse dynamics object

get_vertex_executable_suffix ()
Get the executable suffix for a vertex for this dynamics

get_weight_maximum (*connector*, *weights*)
Get the maximum weight for the synapses

get_weight_mean (*connector*, *weights*)
Get the mean weight for the synapses

get_weight_variance (*connector*, *weights*)
Get the variance in weight for the synapses

get_words (*rows*)
Convert the row data to words

is_same_as (*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

write_parameters (*spec*, *region*, *machine_time_step*, *weight_scales*)
Write the synapse parameters to the spec

spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics_structural module

class spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics_structural.**Ab**
Bases: object

spynnaker.pyNN.models.neuron.synapse_dynamics.pynn_synapse_dynamics module

class spynnaker.pyNN.models.neuron.synapse_dynamics.pynn_synapse_dynamics.**PyNNSynapseDynam**

Bases: object

slow

spynnaker.pyNN.models.neuron.synapse_dynamics.structural_dynamics module

class spynnaker.pyNN.models.neuron.synapse_dynamics.structural_dynamics.**StructuralDynamics**

Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.pynn_synapse_dynamics.PyNNSynapseDynamics*

structure

spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_static module

class spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_static.**SynapseDynamics**

Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_static_synapse_dynamics.AbstractStaticSynapseDynamics*, *spynnaker.pyNN.models.abstract_models.abstract_settable.AbstractSettable*, spinn_front_end_common.abstract_models.abstract_changable_after_run.AbstractChangableAfterRun, *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_generate_on_machine.AbstractGenerateOnMachine*

are_weights_signed()
Determines if the weights are signed values

changes_during_run
Determine if the synapses change during a run

Return type bool

gen_matrix_id
The ID of the on-machine matrix generator

Return type int

get_max_synapses(*n_words*)
Get the maximum number of synapses that can be held in the given number of words

Parameters **n_words** – The number of words the synapses must fit in

Return type int

get_n_static_words_per_row(*ff_size*)
Get the number of bytes to be read per row for the static data given the size that was written to each row

get_n_synapses_in_rows(*ff_size*)
Get the number of synapses in the rows with sizes ff_size

get_n_words_for_static_connections(*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_neurons*, *n_synapse_types*)
Get the SDRAM usage of the synapse dynamics parameters in bytes

get_static_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*)
Get the fixed-fixed data for each row, and lengths for the fixed-fixed parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row for the fixed-fixed region. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for the fixed-fixed region.

get_value(*key*)
Get a property Get a property

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

mark_no_changes()
Marks the point after which changes are reported, so that new changes can be detected before the next check. Marks the point after which changes are reported. Immediately after calling this method, requires_mapping should return False.

read_static_synaptic_data(*post_vertex_slice*, *n_synapse_types*, *ff_size*, *ff_data*)
Read the connections from the words of data in ff_data

requires_mapping()
True if changes that have been made require that mapping be performed. By default this returns False but can be overridden to indicate changes that require mapping.

Return type bool True if changes that have been made require that mapping be performed. Note that this should return True the first time it is called, as the vertex must require mapping as it has been created!

set_value(*key*, *value*)
Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign Set a property
- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

write_parameters(*spec*, *region*, *machine_time_step*, *weight_scales*)
Write the synapse parameters to the spec

spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_stdp module

class spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_stdp.**SynapseDynamicsST**

Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_plastic_synapse_dynamics.AbstractPlasticSynapseDynamics*, *spynnaker.pyNN.models.abstract_models.abstract_settable.AbstractSettable*, spinn_front_end_common.abstract_models.abstract_changable_after_run.AbstractChangableAfterRun, *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_generate_on_machine.AbstractGenerateOnMachine*

are_weights_signed()
Determines if the weights are signed values

changes_during_run
Determine if the synapses change during a run

Return type bool

dendritic_delay_fraction

gen_matrix_id
The ID of the on-machine matrix generator

Return type int

gen_matrix_params
Any parameters required by the matrix generator

Return type numpy array of uint32

gen_matrix_params_size_in_bytes
The size of the parameters of the matrix generator in bytes

Return type int

get_max_synapses(*n_words*)
Get the maximum number of synapses that can be held in the given number of words

Parameters **n_words** – The number of words the synapses must fit in

Return type int

get_n_fixed_plastic_words_per_row(*fp_size*)
Get the number of fixed plastic words to be read from each row

get_n_plastic_plastic_words_per_row(*pp_size*)
Get the number of plastic plastic words to be read from each row

get_n_synapses_in_rows(*pp_size*, *fp_size*)
Get the number of synapses in each of the rows with plastic sizes pp_size and fp_size

get_n_words_for_plastic_connections(*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_neurons*, *n_synapse_types*)
Get the SDRAM usage of the synapse dynamics parameters in bytes

get_plastic_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*)
Get the fixed-plastic data, and plastic-plastic data for each row, and lengths for the fixed_plastic and plastic-plastic parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row, for each of the fixed-plastic and plastic-plastic data regions. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for each of the fixed-plastic and plastic-plastic regions.

get_provenance_data(*pre_population_label*, *post_population_label*)
Get the provenance data from this synapse dynamics object

get_value(*key*)
Get a property Get a property

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

get_weight_maximum(*connector*, *weights*)
Get the maximum weight for the synapses

get_weight_mean(*connector*, *weights*)
Get the mean weight for the synapses

get_weight_variance(*connector*, *weights*)
Get the variance in weight for the synapses

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

mark_no_changes()
Marks the point after which changes are reported, so that new changes can be detected before the next check. Marks the point after which changes are reported. Immediately after calling this method, requires_mapping should return False.

read_plastic_synaptic_data(*post_vertex_slice*, *n_synapse_types*, *pp_size*, *pp_data*, *fp_size*, *fp_data*)
Read the connections indicated in the connection indices from the data in pp_data and fp_data

requires_mapping()
True if changes that have been made require that mapping be performed. By default this returns False but can be overridden to indicate changes that require mapping.

Return type bool True if changes that have been made require that mapping be performed. Note that this should return True the first time it is called, as the vertex must require mapping as it has been created!

set_value(*key*, *value*)
Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign Set a property
- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

timing_dependence

weight_dependence

write_parameters(*spec*, *region*, *machine_time_step*, *weight_scales*)
Write the synapse parameters to the spec

spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_common module

class spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_common.**Syna**

Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics_structural.AbstractSynapseDynamicsStructural*

Common class that enables synaptic rewiring. It acts as a wrapper around SynapseDynamicsStatic or SynapseDynamicsSTDP. This means rewiring can operate in parallel with these types of synapses.

Written by Petrut Bogdan.

Parameters

- **f_rew** (*int*) – Frequency of rewiring (Hz). How many rewiring attempts will be done per second.
- **weight** (*float*) – Initial weight assigned to a newly formed connection
- **delay** (*int*) – Delay assigned to a newly formed connection
- **s_max** (*int*) – Maximum fan-in per target layer neuron
- **sigma_form_forward** (*float*) – Spread of feed-forward formation receptive field
- **sigma_form_lateral** (*float*) – Spread of lateral formation receptive field
- **p_form_forward** (*float*) – Peak probability for feed-forward formation
- **p_form_lateral** (*float*) – Peak probability for lateral formation
- **p_elim_pot** (*float*) – Probability of elimination of a potentiated synapse
- **p_elim_dep** (*float*) – Probability of elimination of a depressed synapse
- **grid** (*2d int array*) – Grid shape
- **lateral_inhibition** (*bool*) – Flag whether to mark synapses formed within a layer as inhibitory or excitatory
- **random_partner** (*bool*) – Flag whether to randomly select pre-synaptic partner for formation
- **seed** (*int*) – seed the random number generators

actual_sdram_usage
Actual SDRAM usage (based on what is written to spec).

Returns actual SDRAM usage

Return type int

default_parameters = {'delay': 1, 'f_rew': 10000, 'grid': array([16, 16]), 'lateral_

distance (*x0*, *x1*, *grid=array([16, 16])*, *type='euclidian'*)
Compute the distance between points x0 and x1 place on the grid using periodic boundary conditions.

Parameters

- **x0** (*np.ndarray of ints*) – first point in space
- **x1** (*np.ndarray of ints*) – second point in space
- **grid** (*np.ndarray of ints*) – shape of grid
- **type** (*str*) – distance metric, i.e. euclidian or manhattan

Returns the distance

Return type float

generate_distance_probability_array (*probability*, *sigma*)
Generate the exponentially decaying probability LUTs.

Parameters

- **probability** (*float*) – peak probability
- **sigma** (*float*) – spread

Returns distance-dependent probabilities

Return type numpy.ndarray(float)

get_extra_sdram_usage_in_bytes (*machine_in_edges*)
Better approximation of SDRAM usage based on incoming machine edges

Parameters **machine_in_edges** (*machine edges*) – incoming machine edges

Returns SDRAM usage

Return type int

get_n_synapses_in_rows (*pp_size*, *fp_size=None*)
Get number of synapses in a row.

get_parameter_names ()

get_parameters_sdram_usage_in_bytes (*n_neurons*, *n_synapse_types*, *in_edges*)
Approximate SDRAM usage

Parameters

- **n_neurons** (*int*) – number of neurons
- **n_synapse_types** (*int*) – number of synapse types (i.e. excitatory and inhibitory)
- **in_edges** (*edges*) – incoming edges

Returns SDRAM usage

Return type int

get_vertex_executable_suffix ()

is_same_as(*synapse_dynamics*)

n_words_for_plastic_connections(*value*)
Get size of plastic connections in words

n_words_for_static_connections(*value*)
Get size of static connections in words

p_rew
The period of rewiring.

Returns The period of rewiring

Return type int

synaptic_data_update(*connections*, *post_vertex_slice*, *app_edge*, *machine_edge*)
Get static synaptic data

weight_dynamics

write_parameters(*spec*, *region*, *machine_time_step*, *weight_scales*, *application_graph*, *machine_graph*, *app_vertex*, *post_slice*, *machine_vertex*, *graph_mapper*, *routing_info*)
Write the synapse parameters to the spec.

Parameters

- **spec** (*spec*) – the data spec
- **region** (*int*) – memory region
- **machine_time_step** (*int*) – the duration of a machine time step (ms)
- **weight_scales** (*list(float)*) – scaling the weights
- **application_graph** (ApplicationGraph) – the entire, highest level, graph of the network to be simulated
- **machine_graph** (MachineGraph) – the entire, lowest level, graph of the network to be simulated
- **app_vertex** (ApplicationVertex) – the highest level object of the post-synaptic population
- **post_slice** (Slice) – the slice of the app vertex corresponding to this machine vertex
- **machine_vertex** (MachineVertex) – the lowest level object of the post-synaptic population
- **graph_mapper** (GraphMapper) – for looking up application vertices
- **routing_info** (RoutingInfo) – All of the routing information on the network

Returns None

Return type None

spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_static module

class spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_static.**Syna**

Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics_structural.AbstractSynapseDynamicsStructural*, *spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_static.SynapseDynamicsStatic*

Class that enables synaptic rewiring. It acts as a wrapper around SynapseDynamicsStatic. This means rewiring can operate in parallel with these types of synapses.

Written by Petrut Bogdan.

Example usage to allow rewiring in parallel with STDP:

```
stdp_model = sim.STDPMechanism(...)

structure_model_with_stdp = sim.StructuralMechanismStatic(
    weight=0,
    s_max=32,
    grid=[np.sqrt(pop_size), np.sqrt(pop_size)],
    random_partner=True,
    f_rew=10 ** 4,  # Hz
    sigma_form_forward=1.,
    delay=10
)
plastic_projection = sim.Projection(
    ...,
    synapse_dynamics=sim.SynapseDynamics(
        slow=structure_model_with_stdp
    )
)
```

Parameters

- **f_rew** (*int*) – Frequency of rewiring (Hz). How many rewiring attempts will be done per second.
- **weight** (*float*) – Initial weight assigned to a newly formed connection

- **delay** (*int*) – Delay assigned to a newly formed connection
- **s_max** (*int*) – Maximum fan-in per target layer neuron
- **sigma_form_forward** (*float*) – Spread of feed-forward formation receptive field
- **sigma_form_lateral** (*float*) – Spread of lateral formation receptive field
- **p_form_forward** (*float*) – Peak probability for feed-forward formation
- **p_form_lateral** (*float*) – Peak probability for lateral formation
- **p_elim_pot** (*float*) – Probability of elimination of a potentiated synapse
- **p_elim_dep** (*float*) – Probability of elimination of a depressed synapse
- **grid** (*2d int array*) – Grid shape
- **lateral_inhibition** (*bool*) – Flag whether to mark synapses formed within a layer as inhibitory or excitatory
- **random_partner** (*bool*) – Flag whether to randomly select pre-synaptic partner for formation
- **seed** (*int*) – seed the random number generators

changes_during_run
Determine if the synapses change during a run

Return type bool

get_n_words_for_static_connections(*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_neurons*, *n_synapse_types*, *in_edges*)
Get the SDRAM usage of the synapse dynamics parameters in bytes

get_static_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*, *app_edge*, *machine_edge*)
Get the fixed-fixed data for each row, and lengths for the fixed-fixed parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row for the fixed-fixed region. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for the fixed-fixed region.

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

write_parameters(*spec*, *region*, *machine_time_step*, *weight_scales*, *application_graph*, *machine_graph*, *app_vertex*, *post_slice*, *machine_vertex*, *graph_mapper*, *routing_info*)
Write the synapse parameters to the spec

spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_stdp module

class spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_stdp.**Synaps**

Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics_structural.AbstractSynapseDynamicsStructural*, *spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_stdp.SynapseDynamicsSTDP*

Class that enables synaptic rewiring. It acts as a wrapper around SynapseDynamicsSTDP. This means rewiring can operate in parallel with these types of synapses.

Written by Petrut Bogdan.

Example usage to allow rewiring in parallel with STDP:

```
stdp_model = sim.STDPMechanism(...)

structure_model_with_stdp = sim.StructuralMechanismSTDP(
    stdp_model=stdp_model,
    weight=0,
    s_max=32,
    grid=[np.sqrt(pop_size), np.sqrt(pop_size)],
    random_partner=True,
    f_rew=10 ** 4,  # Hz
    sigma_form_forward=1.,
    delay=10
)
plastic_projection = sim.Projection(
    ...,
    synapse_dynamics=sim.SynapseDynamics(
        slow=structure_model_with_stdp
    )
)
```

Parameters

- **f_rew** (*int*) – Frequency of rewiring (Hz). How many rewiring attempts will be done per second.
- **weight** (*float*) – Initial weight assigned to a newly formed connection

- **delay** (*int*) – Delay assigned to a newly formed connection
- **s_max** (*int*) – Maximum fan-in per target layer neuron
- **sigma_form_forward** (*float*) – Spread of feed-forward formation receptive field
- **sigma_form_lateral** (*float*) – Spread of lateral formation receptive field
- **p_form_forward** (*float*) – Peak probability for feed-forward formation
- **p_form_lateral** (*float*) – Peak probability for lateral formation
- **p_elim_pot** (*float*) – Probability of elimination of a potentiated synapse
- **p_elim_dep** (*float*) – Probability of elimination of a depressed synapse
- **grid** (*2d int array*) – Grid shape
- **lateral_inhibition** (*bool*) – Flag whether to mark synapses formed within a layer as inhibitory or excitatory
- **random_partner** (*bool*) – Flag whether to randomly select pre-synaptic partner for formation
- **seed** (*int*) – seed the random number generators

get_n_words_for_plastic_connections(*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_neurons*, *n_synapse_types*, *in_edges*)
Get the SDRAM usage of the synapse dynamics parameters in bytes

get_plastic_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*, *app_edge*, *machine_edge*)
Get the fixed-plastic data, and plastic-plastic data for each row, and lengths for the fixed_plastic and plastic-plastic parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row, for each of the fixed-plastic and plastic-plastic data regions. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for each of the fixed-plastic and plastic-plastic regions.

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

write_parameters(*spec*, *region*, *machine_time_step*, *weight_scales*, *application_graph*, *machine_graph*, *app_vertex*, *post_slice*, *machine_vertex*, *graph_mapper*, *routing_info*)
Write the synapse parameters to the spec

Module contents

class spynnaker.pyNN.models.neuron.synapse_dynamics.**AbstractSynapseDynamics**
Bases: object

NUMPY_CONNECTORS_DTYPE = [('source', 'uint32'), ('target', 'uint32'), ('weight', 'floa

are_weights_signed()
Determines if the weights are signed values

changes_during_run
Determine if the synapses change during a run

Return type bool

convert_per_connection_data_to_rows(*connection_row_indices*, *n_rows*, *data*)
Converts per-connection data generated from connections into row-based data to be returned from get_synaptic_data

get_delay_maximum(*connector*, *delays*)
Get the maximum delay for the synapses

get_delay_variance(*connector*, *delays*)
Get the variance in delay for the synapses

get_max_synapses(*n_words*)
Get the maximum number of synapses that can be held in the given number of words

Parameters **n_words** – The number of words the synapses must fit in

Return type int

get_n_items(*rows*, *item_size*)
Get the number of items in each row as 4-byte values, given the item size

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_neurons*, *n_synapse_types*)
Get the SDRAM usage of the synapse dynamics parameters in bytes

get_provenance_data(*pre_population_label*, *post_population_label*)
Get the provenance data from this synapse dynamics object

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

get_weight_maximum(*connector*, *weights*)
Get the maximum weight for the synapses

get_weight_mean(*connector*, *weights*)
Get the mean weight for the synapses

get_weight_variance(*connector*, *weights*)
Get the variance in weight for the synapses

get_words(*rows*)
Convert the row data to words

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

write_parameters (*spec*, *region*, *machine_time_step*, *weight_scales*)
Write the synapse parameters to the spec

class spynnaker.pyNN.models.neuron.synapse_dynamics.**AbstractGenerateOnMachine**
Bases: object

A synapse dynamics that can be generated on the machine

gen_matrix_id
The ID of the on-machine matrix generator

Return type int

gen_matrix_params
Any parameters required by the matrix generator

Return type numpy array of uint32

gen_matrix_params_size_in_bytes
The size of the parameters of the matrix generator in bytes

Return type int

generate_on_machine ()
Determines if this instance should be generated on the machine.

Default implementation returns True

Return type bool

class spynnaker.pyNN.models.neuron.synapse_dynamics.**AbstractStaticSynapseDynamics**
Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics.AbstractSynapseDynamics*

Dynamics which don't change over time.

get_n_static_words_per_row (*ff_size*)
Get the number of bytes to be read per row for the static data given the size that was written to each row

get_n_synapses_in_rows (*ff_size*)
Get the number of synapses in the rows with sizes ff_size

get_n_words_for_static_connections (*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_static_synaptic_data (*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*)
Get the fixed-fixed data for each row, and lengths for the fixed-fixed parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row for the fixed-fixed region. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for the fixed-fixed region.

read_static_synaptic_data (*post_vertex_slice*, *n_synapse_types*, *ff_size*, *ff_data*)
Read the connections from the words of data in ff_data

class spynnaker.pyNN.models.neuron.synapse_dynamics.**AbstractPlasticSynapseDynamics**
Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics.AbstractSynapseDynamics*

Synapses which change over time

get_n_fixed_plastic_words_per_row (*fp_size*)
Get the number of fixed plastic words to be read from each row

get_n_plastic_plastic_words_per_row(*pp_size*)
Get the number of plastic plastic words to be read from each row

get_n_synapses_in_rows(*pp_size*, *fp_size*)
Get the number of synapses in each of the rows with plastic sizes pp_size and fp_size

get_n_words_for_plastic_connections(*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_plastic_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*)
Get the fixed-plastic data, and plastic-plastic data for each row, and lengths for the fixed_plastic and plastic-plastic parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row, for each of the fixed-plastic and plastic-plastic data regions. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for each of the fixed-plastic and plastic-plastic regions.

read_plastic_synaptic_data(*post_vertex_slice*, *n_synapse_types*, *pp_size*, *pp_data*, *fp_size*, *fp_data*)
Read the connections indicated in the connection indices from the data in pp_data and fp_data

class `spynnaker.pyNN.models.neuron.synapse_dynamics.`**PyNNSynapseDynamics**(*slow=None*, *fast=None*)
Bases: `object`

slow

class `spynnaker.pyNN.models.neuron.synapse_dynamics.`**SynapseDynamicsStatic**(*pad_to_length=None*)
Bases: `spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_static_synapse_dynamics.AbstractStaticSynapseDynamics`, `spynnaker.pyNN.models.abstract_models.abstract_settable.AbstractSettable`, `spinn_front_end_common.abstract_models.abstract_changable_after_run.AbstractChangableAfterRun`, `spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_generate_on_machine.AbstractGenerateOnMachine`

are_weights_signed()
Determines if the weights are signed values

changes_during_run
Determine if the synapses change during a run

Return type bool

gen_matrix_id
The ID of the on-machine matrix generator

Return type int

get_max_synapses(*n_words*)
Get the maximum number of synapses that can be held in the given number of words

Parameters **n_words** – The number of words the synapses must fit in

Return type int

get_n_static_words_per_row(*ff_size*)
Get the number of bytes to be read per row for the static data given the size that was written to each row

get_n_synapses_in_rows(*ff_size*)
Get the number of synapses in the rows with sizes ff_size

get_n_words_for_static_connections(*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_neurons*, *n_synapse_types*)
Get the SDRAM usage of the synapse dynamics parameters in bytes

get_static_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*)
Get the fixed-fixed data for each row, and lengths for the fixed-fixed parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row for the fixed-fixed region. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for the fixed-fixed region.

get_value(*key*)
Get a property Get a property

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

mark_no_changes()
Marks the point after which changes are reported, so that new changes can be detected before the next check. Marks the point after which changes are reported. Immediately after calling this method, requires_mapping should return False.

read_static_synaptic_data(*post_vertex_slice*, *n_synapse_types*, *ff_size*, *ff_data*)
Read the connections from the words of data in ff_data

requires_mapping()
True if changes that have been made require that mapping be performed. By default this returns False but can be overridden to indicate changes that require mapping.

Return type bool True if changes that have been made require that mapping be performed. Note that this should return True the first time it is called, as the vertex must require mapping as it has been created!

set_value(*key*, *value*)
Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign Set a property
- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

write_parameters(*spec*, *region*, *machine_time_step*, *weight_scales*)
Write the synapse parameters to the spec

class spynnaker.pyNN.models.neuron.synapse_dynamics.**SynapseDynamicsSTDP**(*timing_dependence=None, weight_dependence=None, voltage_dependence=None, dendritic_delay_fraction=1.0, pad_to_length=None*)

Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_plastic_synapse_dynamics.AbstractPlasticSynapseDynamics*, *spynnaker.pyNN.models.abstract_models.abstract_settable.AbstractSettable*, spinn_front_end_common.abstract_models.abstract_changable_after_run.AbstractChangableAfterRun, *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_generate_on_machine.AbstractGenerateOnMachine*

are_weights_signed()
Determines if the weights are signed values

changes_during_run
Determine if the synapses change during a run

Return type bool

dendritic_delay_fraction

gen_matrix_id
The ID of the on-machine matrix generator

Return type int

gen_matrix_params
Any parameters required by the matrix generator

Return type numpy array of uint32

gen_matrix_params_size_in_bytes
The size of the parameters of the matrix generator in bytes

Return type int

get_max_synapses(*n_words*)
Get the maximum number of synapses that can be held in the given number of words

Parameters **n_words** – The number of words the synapses must fit in

Return type int

get_n_fixed_plastic_words_per_row(*fp_size*)
Get the number of fixed plastic words to be read from each row

get_n_plastic_plastic_words_per_row(*pp_size*)
Get the number of plastic plastic words to be read from each row

get_n_synapses_in_rows(*pp_size, fp_size*)
Get the number of synapses in each of the rows with plastic sizes pp_size and fp_size

get_n_words_for_plastic_connections(*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_parameters_sdram_usage_in_bytes (*n_neurons*, *n_synapse_types*)
Get the SDRAM usage of the synapse dynamics parameters in bytes

get_plastic_synaptic_data (*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*)
Get the fixed-plastic data, and plastic-plastic data for each row, and lengths for the fixed_plastic and plastic-plastic parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row, for each of the fixed-plastic and plastic-plastic data regions. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for each of the fixed-plastic and plastic-plastic regions.

get_provenance_data (*pre_population_label*, *post_population_label*)
Get the provenance data from this synapse dynamics object

get_value (*key*)
Get a property Get a property

get_vertex_executable_suffix ()
Get the executable suffix for a vertex for this dynamics

get_weight_maximum (*connector*, *weights*)
Get the maximum weight for the synapses

get_weight_mean (*connector*, *weights*)
Get the mean weight for the synapses

get_weight_variance (*connector*, *weights*)
Get the variance in weight for the synapses

is_same_as (*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

mark_no_changes ()
Marks the point after which changes are reported, so that new changes can be detected before the next check. Marks the point after which changes are reported. Immediately after calling this method, requires_mapping should return False.

read_plastic_synaptic_data (*post_vertex_slice*, *n_synapse_types*, *pp_size*, *pp_data*, *fp_size*, *fp_data*)
Read the connections indicated in the connection indices from the data in pp_data and fp_data

requires_mapping ()
True if changes that have been made require that mapping be performed. By default this returns False but can be overridden to indicate changes that require mapping.

Return type bool True if changes that have been made require that mapping be performed. Note that this should return True the first time it is called, as the vertex must require mapping as it has been created!

set_value (*key*, *value*)
Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign Set a property
- **key** – the name of the parameter to change

- **value** – the new value of the parameter to assign

timing_dependence

weight_dependence

write_parameters (*spec*, *region*, *machine_time_step*, *weight_scales*)
Write the synapse parameters to the spec

class spynnaker.pyNN.models.neuron.synapse_dynamics.**AbstractSynapseDynamicsStructural**
Bases: object

class spynnaker.pyNN.models.neuron.synapse_dynamics.**StructuralDynamics** (*slow=None*, *fast=None*, *structure=None*)
Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.pynn_synapse_dynamics.PyNNSynapseDynamics*

structure

class spynnaker.pyNN.models.neuron.synapse_dynamics.**SynapseDynamicsStructuralCommon** (*stdp_mod* *f_rew=10* *weight=0* *delay=1*, *s_max=3* *sigma_fo* *sigma_fo* *p_form_f* *p_form_l* *p_elim_d* *p_elim_p* *grid=arra* *16])*, *lateral_inhib* *random_part* *seed=Nor*
Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics_structural.AbstractSynapseDynamicsStructural*

Common class that enables synaptic rewiring. It acts as a wrapper around SynapseDynamicsStatic or SynapseDynamicsSTDP. This means rewiring can operate in parallel with these types of synapses.

Written by Petrut Bogdan.

Parameters

- **f_rew** (*int*) – Frequency of rewiring (Hz). How many rewiring attempts will be done per second.
- **weight** (*float*) – Initial weight assigned to a newly formed connection
- **delay** (*int*) – Delay assigned to a newly formed connection
- **s_max** (*int*) – Maximum fan-in per target layer neuron
- **sigma_form_forward** (*float*) – Spread of feed-forward formation receptive field
- **sigma_form_lateral** (*float*) – Spread of lateral formation receptive field

- **p_form_forward** (*float*) – Peak probability for feed-forward formation
- **p_form_lateral** (*float*) – Peak probability for lateral formation
- **p_elim_pot** (*float*) – Probability of elimination of a potentiated synapse
- **p_elim_dep** (*float*) – Probability of elimination of a depressed synapse
- **grid** (*2d int array*) – Grid shape
- **lateral_inhibition** (*bool*) – Flag whether to mark synapses formed within a layer as inhibitory or excitatory
- **random_partner** (*bool*) – Flag whether to randomly select pre-synaptic partner for formation
- **seed** (*int*) – seed the random number generators

actual_sdram_usage
Actual SDRAM usage (based on what is written to spec).

Returns actual SDRAM usage

Return type int

default_parameters = {'delay': 1, 'f_rew': 10000, 'grid': array([16, 16]), 'lateral_

distance (*x0*, *x1*, *grid=array([16, 16])*, *type='euclidian'*)
Compute the distance between points x0 and x1 place on the grid using periodic boundary conditions.

Parameters

- **x0** (*np.ndarray of ints*) – first point in space
- **x1** (*np.ndarray of ints*) – second point in space
- **grid** (*np.ndarray of ints*) – shape of grid
- **type** (*str*) – distance metric, i.e. euclidian or manhattan

Returns the distance

Return type float

generate_distance_probability_array (*probability*, *sigma*)
Generate the exponentially decaying probability LUTs.

Parameters

- **probability** (*float*) – peak probability
- **sigma** (*float*) – spread

Returns distance-dependent probabilities

Return type numpy.ndarray(float)

get_extra_sdram_usage_in_bytes (*machine_in_edges*)
Better approximation of SDRAM usage based on incoming machine edges

Parameters **machine_in_edges** (*machine edges*) – incoming machine edges

Returns SDRAM usage

Return type int

get_n_synapses_in_rows (*pp_size*, *fp_size=None*)
Get number of synapses in a row.

get_parameter_names()

get_parameters_sdram_usage_in_bytes(*n_neurons*, *n_synapse_types*, *in_edges*)
Approximate SDRAM usage

Parameters

- **n_neurons** (`int`) – number of neurons
- **n_synapse_types** (`int`) – number of synapse types (i.e. excitatory and inhibitory)
- **in_edges** (`edges`) – incoming edges

Returns SDRAM usage

Return type int

get_vertex_executable_suffix()

is_same_as(*synapse_dynamics*)

n_words_for_plastic_connections(*value*)
Get size of plastic connections in words

n_words_for_static_connections(*value*)
Get size of static connections in words

p_rew
The period of rewiring.

Returns The period of rewiring

Return type int

synaptic_data_update(*connections*, *post_vertex_slice*, *app_edge*, *machine_edge*)
Get static synaptic data

weight_dynamics

write_parameters(*spec*, *region*, *machine_time_step*, *weight_scales*, *application_graph*, *machine_graph*, *app_vertex*, *post_slice*, *machine_vertex*, *graph_mapper*, *routing_info*)
Write the synapse parameters to the spec.

Parameters

- **spec** (`spec`) – the data spec
- **region** (`int`) – memory region
- **machine_time_step** (`int`) – the duration of a machine time step (ms)
- **weight_scales** (`list(float)`) – scaling the weights
- **application_graph** (`ApplicationGraph`) – the entire, highest level, graph of the network to be simulated
- **machine_graph** (`MachineGraph`) – the entire, lowest level, graph of the network to be simulated
- **app_vertex** (`ApplicationVertex`) – the highest level object of the post-synaptic population
- **post_slice** (`Slice`) – the slice of the app vertex corresponding to this machine vertex
- **machine_vertex** (`MachineVertex`) – the lowest level object of the post-synaptic population

- **graph_mapper** (GraphMapper) – for looking up application vertices
- **routing_info** (RoutingInfo) – All of the routing information on the network

Returns None

Return type None

class spynnaker.pyNN.models.neuron.synapse_dynamics.**SynapseDynamicsStructuralStatic**(*stdp_mod f_rew=10 weight=0 de- lay=1, s_max=3 sigma_fo sigma_fo p_form_f p_form_l p_elim_d p_elim_p grid=arr 16]), lat- eral_inhi ran- dom_part seed=Nor*

Bases: *spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics_structural.AbstractSynapseDynamicsStructural*, *spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_static.SynapseDynamicsStatic*

Class that enables synaptic rewiring. It acts as a wrapper around SynapseDynamicsStatic. This means rewiring can operate in parallel with these types of synapses.

Written by Petrut Bogdan.

Example usage to allow rewiring in parallel with STDP:

```
stdp_model = sim.STDPMechanism(...)

structure_model_with_stdp = sim.StructuralMechanismStatic(
    weight=0,
    s_max=32,
    grid=[np.sqrt(pop_size), np.sqrt(pop_size)],
    random_partner=True,
    f_rew=10 ** 4,  # Hz
    sigma_form_forward=1.,
    delay=10
)
plastic_projection = sim.Projection(
    ...,
    synapse_dynamics=sim.SynapseDynamics(
        slow=structure_model_with_stdp
    )
)
```

Parameters

- **f_rew** (*int*) – Frequency of rewiring (Hz). How many rewiring attempts will be done per second.
- **weight** (*float*) – Initial weight assigned to a newly formed connection
- **delay** (*int*) – Delay assigned to a newly formed connection
- **s_max** (*int*) – Maximum fan-in per target layer neuron
- **sigma_form_forward** (*float*) – Spread of feed-forward formation receptive field
- **sigma_form_lateral** (*float*) – Spread of lateral formation receptive field
- **p_form_forward** (*float*) – Peak probability for feed-forward formation
- **p_form_lateral** (*float*) – Peak probability for lateral formation
- **p_elim_pot** (*float*) – Probability of elimination of a potentiated synapse
- **p_elim_dep** (*float*) – Probability of elimination of a depressed synapse
- **grid** (*2d int array*) – Grid shape
- **lateral_inhibition** (*bool*) – Flag whether to mark synapses formed within a layer as inhibitory or excitatory
- **random_partner** (*bool*) – Flag whether to randomly select pre-synaptic partner for formation
- **seed** (*int*) – seed the random number generators

changes_during_run
Determine if the synapses change during a run

Return type bool

get_n_words_for_static_connections(*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_neurons*, *n_synapse_types*, *in_edges*)
Get the SDRAM usage of the synapse dynamics parameters in bytes

get_static_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*, *app_edge*, *machine_edge*)
Get the fixed-fixed data for each row, and lengths for the fixed-fixed parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row for the fixed-fixed region. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for the fixed-fixed region.

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

write_parameters(*spec*, *region*, *machine_time_step*, *weight_scales*, *application_graph*, *machine_graph*, *app_vertex*, *post_slice*, *machine_vertex*, *graph_mapper*, *routing_info*)
Write the synapse parameters to the spec

class spynnaker.pyNN.models.neuron.synapse_dynamics.**SynapseDynamicsStructuralSTDP**(*stdp_model=* *f_rew=1000* *weight=0*, *delay=1*, *s_max=32*, *sigma_form_* *sigma_form_* *p_form_forw* *p_form_late* *p_elim_dep=* *p_elim_pot=* *grid=array([* *16])*, *lateral_inhibiti* *random_partne* *seed=None*)

Bases: `spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics_structural.AbstractSynapseDynamicsStructural`, `spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_stdp.SynapseDynamicsSTDP`

Class that enables synaptic rewiring. It acts as a wrapper around SynapseDynamicsSTDP. This means rewiring can operate in parallel with these types of synapses.

Written by Petrut Bogdan.

Example usage to allow rewiring in parallel with STDP:

```
stdp_model = sim.STDPMechanism(...)

structure_model_with_stdp = sim.StructuralMechanismSTDP(
    stdp_model=stdp_model,
    weight=0,
    s_max=32,
    grid=[np.sqrt(pop_size), np.sqrt(pop_size)],
    random_partner=True,
    f_rew=10 ** 4,  # Hz
    sigma_form_forward=1.,
    delay=10
)
plastic_projection = sim.Projection(
    ...,
    synapse_dynamics=sim.SynapseDynamics(
        slow=structure_model_with_stdp
    )
)
```

Parameters

- **f_rew** (*int*) – Frequency of rewiring (Hz). How many rewiring attempts will be done per second.

- **weight** (*float*) – Initial weight assigned to a newly formed connection
- **delay** (*int*) – Delay assigned to a newly formed connection
- **s_max** (*int*) – Maximum fan-in per target layer neuron
- **sigma_form_forward** (*float*) – Spread of feed-forward formation receptive field
- **sigma_form_lateral** (*float*) – Spread of lateral formation receptive field
- **p_form_forward** (*float*) – Peak probability for feed-forward formation
- **p_form_lateral** (*float*) – Peak probability for lateral formation
- **p_elim_pot** (*float*) – Probability of elimination of a potentiated synapse
- **p_elim_dep** (*float*) – Probability of elimination of a depressed synapse
- **grid** (*2d int array*) – Grid shape
- **lateral_inhibition** (*bool*) – Flag whether to mark synapses formed within a layer as inhibitory or excitatory
- **random_partner** (*bool*) – Flag whether to randomly select pre-synaptic partner for formation
- **seed** (*int*) – seed the random number generators

get_n_words_for_plastic_connections(*n_connections*)
Get the number of 32-bit words for n_connections in a single row

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_parameters_sdram_usage_in_bytes(*n_neurons*, *n_synapse_types*, *in_edges*)
Get the SDRAM usage of the synapse dynamics parameters in bytes

get_plastic_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*, *app_edge*, *machine_edge*)
Get the fixed-plastic data, and plastic-plastic data for each row, and lengths for the fixed_plastic and plastic-plastic parts of each row.

Data is returned as an array made up of an array of 32-bit words for each row, for each of the fixed-plastic and plastic-plastic data regions. The row into which connection should go is given by connection_row_indices, and the total number of rows is given by n_rows.

Lengths are returned as an array made up of an integer for each row, for each of the fixed-plastic and plastic-plastic regions.

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

write_parameters(*spec*, *region*, *machine_time_step*, *weight_scales*, *application_graph*, *machine_graph*, *app_vertex*, *post_slice*, *machine_vertex*, *graph_mapper*, *routing_info*)
Write the synapse parameters to the spec

spynnaker.pyNN.models.neuron.synapse_io package

Submodules

spynnaker.pyNN.models.neuron.synapse_io.abstract_synapse_io module

class spynnaker.pyNN.models.neuron.synapse_io.abstract_synapse_io.**AbstractSynapseIO**

Bases: object

get_block_n_bytes (*max_row_length*, *n_rows*)

Get the number of bytes in a block given the max row length and number of rows

get_max_row_info (*synapse_info*, *post_vertex_slice*, *n_delay_stages*, *population_table*, *machine_time_step*, *in_edge*)

Get the information about the maximum lengths of delayed and undelayed rows in bytes (including header), words (without header) and number of synapses

get_maximum_delay_supported_in_ms (*machine_time_step*)

Get the maximum delay supported by the synapse representation before extensions are required, or None if any delay is supported

get_synapses (*synapse_info*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *n_delay_stages*, *population_table*, *n_synapse_types*, *weight_scales*, *machine_time_step*, *app_edge*, *machine_edge*)

Get the synapses as an array of words for non-delayed synapses and an array of words for delayed synapses

read_synapses (*synapse_info*, *pre_vertex_slice*, *post_vertex_slice*, *max_row_length*, *delayed_max_row_length*, *n_synapse_types*, *weight_scales*, *data*, *delayed_data*, *n_delay_stages*, *machine_time_step*)

Read the synapses for a given projection synapse information object out of the given data

spynnaker.pyNN.models.neuron.synapse_io.max_row_info module

class spynnaker.pyNN.models.neuron.synapse_io.max_row_info.**MaxRowInfo** (*undelayed_max_n_synapses*, *delayed_max_n_synapses*, *undelayed_max_bytes*, *delayed_max_bytes*, *undelayed_max_words*, *delayed_max_words*)

Bases: object

Information about the maximums for rows in a synaptic matrix.

delayed_max_bytes

delayed_max_n_synapses

delayed_max_words

undelayed_max_bytes

undelayed_max_n_synapses

undelayed_max_words

spynnaker.pyNN.models.neuron.synapse_io.synapse_io_row_based module

class spynnaker.pyNN.models.neuron.synapse_io.synapse_io_row_based.**SynapseIORowBased**
Bases: *spynnaker.pyNN.models.neuron.synapse_io.abstract_synapse_io.AbstractSynapseIO*

A SynapseRowIO implementation that uses a row for each source neuron, where each row consists of a fixed region, a plastic region, and a fixed-plastic region (this is the bits of the plastic row that don't actually change). The plastic region structure is determined by the synapse dynamics of the connector.

get_block_n_bytes(*max_row_length, n_rows*)
Get the number of bytes in a block given the max row length and number of rows

get_max_row_info(*synapse_info, post_vertex_slice, n_delay_stages, population_table, machine_time_step, in_edge*)
Get the information about the maximum lengths of delayed and undelayed rows in bytes (including header), words (without header) and number of synapses

get_maximum_delay_supported_in_ms(*machine_time_step*)
Get the maximum delay supported by the synapse representation before extensions are required, or None if any delay is supported

get_synapses(*synapse_info, pre_slices, pre_slice_index, post_slices, post_slice_index, pre_vertex_slice, post_vertex_slice, n_delay_stages, population_table, n_synapse_types, weight_scales, machine_time_step, app_edge, machine_edge*)
Get the synapses as an array of words for non-delayed synapses and an array of words for delayed synapses

read_synapses(*synapse_info, pre_vertex_slice, post_vertex_slice, max_row_length, delayed_max_row_length, n_synapse_types, weight_scales, data, delayed_data, n_delay_stages, machine_time_step*)
Read the synapses for a given projection synapse information object out of the given data

Module contents

class spynnaker.pyNN.models.neuron.synapse_io.**AbstractSynapseIO**
Bases: object

get_block_n_bytes(*max_row_length, n_rows*)
Get the number of bytes in a block given the max row length and number of rows

get_max_row_info(*synapse_info, post_vertex_slice, n_delay_stages, population_table, machine_time_step, in_edge*)
Get the information about the maximum lengths of delayed and undelayed rows in bytes (including header), words (without header) and number of synapses

get_maximum_delay_supported_in_ms(*machine_time_step*)
Get the maximum delay supported by the synapse representation before extensions are required, or None if any delay is supported

get_synapses(*synapse_info, pre_slices, pre_slice_index, post_slices, post_slice_index, pre_vertex_slice, post_vertex_slice, n_delay_stages, population_table, n_synapse_types, weight_scales, machine_time_step, app_edge, machine_edge*)
Get the synapses as an array of words for non-delayed synapses and an array of words for delayed synapses

read_synapses(*synapse_info, pre_vertex_slice, post_vertex_slice, max_row_length, delayed_max_row_length, n_synapse_types, weight_scales, data, delayed_data, n_delay_stages, machine_time_step*)
Read the synapses for a given projection synapse information object out of the given data

class spynnaker.pyNN.models.neuron.synapse_io.**SynapseIORowBased**
Bases: *spynnaker.pyNN.models.neuron.synapse_io.abstract_synapse_io.AbstractSynapseIO*

A SynapseRowIO implementation that uses a row for each source neuron, where each row consists of a fixed region, a plastic region, and a fixed-plastic region (this is the bits of the plastic row that don't actually change). The plastic region structure is determined by the synapse dynamics of the connector.

get_block_n_bytes(*max_row_length*, *n_rows*)
Get the number of bytes in a block given the max row length and number of rows

get_max_row_info(*synapse_info*, *post_vertex_slice*, *n_delay_stages*, *population_table*, *machine_time_step*, *in_edge*)
Get the information about the maximum lengths of delayed and undelayed rows in bytes (including header), words (without header) and number of synapses

get_maximum_delay_supported_in_ms(*machine_time_step*)
Get the maximum delay supported by the synapse representation before extensions are required, or None if any delay is supported

get_synapses(*synapse_info*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *n_delay_stages*, *population_table*, *n_synapse_types*, *weight_scales*, *machine_time_step*, *app_edge*, *machine_edge*)
Get the synapses as an array of words for non-delayed synapses and an array of words for delayed synapses

read_synapses(*synapse_info*, *pre_vertex_slice*, *post_vertex_slice*, *max_row_length*, *delayed_max_row_length*, *n_synapse_types*, *weight_scales*, *data*, *delayed_data*, *n_delay_stages*, *machine_time_step*)
Read the synapses for a given projection synapse information object out of the given data

spynnaker.pyNN.models.neuron.synapse_types package

Submodules

spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type module

class spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type.**AbstractSynapseType**
Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.AbstractStandardNeuronComponent*

Represents the synapse types supported.

Parameters **data_types** – A list of data types in the component structure, in the order that they appear

get_n_synapse_types()
Get the number of synapse types supported.

Returns The number of synapse types supported

Return type int

get_synapse_id_by_target(*target*)
Get the ID of a synapse given the name.

Returns The ID of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type.

Returns an array of strings

Return type array(str)

spynnaker.pyNN.models.neuron.synapse_types.synapse_type_alpha module

class spynnaker.pyNN.models.neuron.synapse_types.synapse_type_alpha.**SynapseTypeAlpha**(*exc_resp*
exc_exp
tau_syn
inh_resp
inh_exp
tau_syn

Bases: *spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type.AbstractSynapseType*

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters state_variables (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

exc_response

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters n_neurons (*int*) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported.

Returns The number of synapse types supported

Return type int

get_synapse_id_by_target(*target*)
Get the ID of a synapse given the name.

Returns The ID of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type.

Returns an array of strings

Return type array(str)

get_units(*variable*)
Get the units of the given variable

Parameters variable (*str*) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters variable (*str*) – The name of the variable

Return type bool

inh_response

tau_syn_E

tau_syn_I

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

spynnaker.pyNN.models.neuron.synapse_types.synapse_type_delta module

class spynnaker.pyNN.models.neuron.synapse_types.synapse_type_delta.**SynapseTypeDelta** (*isyn_exc*, *isyn_inh*)
Bases: *spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type.AbstractSynapseType*

This represents a synapse type with two delta synapses

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters state_variables (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported.

Returns The number of synapse types supported

Return type int

get_synapse_id_by_target(*target*)
Get the ID of a synapse given the name.

Returns The ID of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type.

Returns an array of strings

Return type array(str)

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (*str*) – The name of the variable

Return type bool

isyn_exc

isyn_inh

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

spynnaker.pyNN.models.neuron.synapse_types.synapse_type_dual_exponential module

class spynnaker.pyNN.models.neuron.synapse_types.synapse_type_dual_exponential.**SynapseTypeI**

Bases: *spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type.AbstractSynapseType*

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters state_variables (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters n_neurons (*int*) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported.

Returns The number of synapse types supported

Return type int

get_synapse_id_by_target(*target*)
Get the ID of a synapse given the name.

Returns The ID of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type.

Returns an array of strings

Return type array(str)

get_units(*variable*)
Get the units of the given variable

Parameters variable (*str*) – The name of the variable

get_values(*parameters, state_variables, vertex_slice, ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters

- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (*str*) – The name of the variable

Return type bool

isyn_exc

isyn_exc2

isyn_inh

tau_syn_E

tau_syn_E2

tau_syn_I

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

spynnaker.pyNN.models.neuron.synapse_types.synapse_type_exponential module

class spynnaker.pyNN.models.neuron.synapse_types.synapse_type_exponential.**SynapseTypeExpone**

Bases: *spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type.AbstractSynapseType*

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported.

Returns The number of synapse types supported

Return type int

get_synapse_id_by_target(*target*)
Get the ID of a synapse given the name.

Returns The ID of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type.

Returns an array of strings

Return type array(str)

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

isyn_exc

isyn_inh

tau_syn_E

tau_syn_I

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update

- **state_variables** – The holder of the state variables to update

Module contents

class spynnaker.pyNN.models.neuron.synapse_types.**AbstractSynapseType**(*data_types*)

Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.AbstractStandardNeuronComponent*

Represents the synapse types supported.

Parameters **data_types** – A list of data types in the component structure, in the order that they appear

get_n_synapse_types()

Get the number of synapse types supported.

Returns The number of synapse types supported

Return type int

get_synapse_id_by_target(*target*)

Get the ID of a synapse given the name.

Returns The ID of the synapse

Return type int

get_synapse_targets()

Get the target names of the synapse type.

Returns an array of strings

Return type array(str)

class spynnaker.pyNN.models.neuron.synapse_types.**SynapseTypeDualExponential**(*tau_syn_E*, *tau_syn_E2*, *tau_syn_I*, *isyn_exc*, *isyn_exc2*, *isyn_inh*)

Bases: *spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type.AbstractSynapseType*

add_parameters(*parameters*)

Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)

Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_n_cpu_cycles(*n_neurons*)

Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported.

Returns The number of synapse types supported

Return type int

get_synapse_id_by_target(*target*)
Get the ID of a synapse given the name.

Returns The ID of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type.

Returns an array of strings

Return type array(str)

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

isyn_exc

isyn_exc2

isyn_inh

tau_syn_E

tau_syn_E2

tau_syn_I

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element

- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

class spynnaker.pyNN.models.neuron.synapse_types.**SynapseTypeExponential**(*tau_syn_E*, *tau_syn_I*, *isyn_exc*, *isyn_inh*)

Bases: *spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type.AbstractSynapseType*

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters state_variables (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters n_neurons (*int*) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported.

Returns The number of synapse types supported

Return type int

get_synapse_id_by_target(*target*)
Get the ID of a synapse given the name.

Returns The ID of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type.

Returns an array of strings

Return type array(str)

get_units(*variable*)
Get the units of the given variable

Parameters variable (*str*) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables

- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

isyn_exc

isyn_inh

tau_syn_E

tau_syn_I

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

class spynnaker.pyNN.models.neuron.synapse_types.**SynapseTypeDelta**(*isyn_exc*, *isyn_inh*)
Bases: *spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type.AbstractSynapseType*

This represents a synapse type with two delta synapses

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the state variables

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (`int`) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types()
Get the number of synapse types supported.

Returns The number of synapse types supported

Return type int

get_synapse_id_by_target(*target*)
Get the ID of a synapse given the name.

Returns The ID of the synapse

Return type int

get_synapse_targets()
Get the target names of the synapse type.

Returns an array of strings

Return type array(str)

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (*str*) – The name of the variable

Return type bool

isyn_exc

isyn_inh

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

class spynnaker.pyNN.models.neuron.synapse_types.**SynapseTypeAlpha**(*exc_response*, *exc_exp_response*, *tau_syn_E*, *inh_response*, *inh_exp_response*, *tau_syn_I*)
Bases: *spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type.AbstractSynapseType*

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

exc_response

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (int) – The number of neurons to get the cycles for

Return type int

get_n_synapse_types ()
Get the number of synapse types supported.

Returns The number of synapse types supported

Return type int

get_synapse_id_by_target (*target*)
Get the ID of a synapse given the name.

Returns The ID of the synapse

Return type int

get_synapse_targets ()
Get the target names of the synapse type.

Returns an array of strings

Return type array(str)

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (str) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the parameters
- **state_variables** (spinn_utilities.ranged.range_dictionary.RangeDictionary) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (str) – The name of the variable

Return type bool

inh_response

tau_syn_E

tau_syn_I

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

spynnaker.pyNN.models.neuron.threshold_types package

Submodules

spynnaker.pyNN.models.neuron.threshold_types.abstract_threshold_type module

class spynnaker.pyNN.models.neuron.threshold_types.abstract_threshold_type.**AbstractThresho**
Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.AbstractStandardNeuronComponent*

Represents types of threshold for a neuron (e.g., stochastic).

Parameters data_types – A list of data types in the component structure, in the order that they appear

spynnaker.pyNN.models.neuron.threshold_types.threshold_type_maass_stochastic module

class spynnaker.pyNN.models.neuron.threshold_types.threshold_type_maass_stochastic.**Thresho**

Bases: *spynnaker.pyNN.models.neuron.threshold_types.abstract_threshold_type.AbstractThresholdType*

A stochastic threshold

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters parameters (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters state_variables (spinn_utilities.ranged.range_dictionary.RangeDictionary) – A holder of the state variables

du_th

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters n_neurons (*int*) – The number of neurons to get the cycles for

Return type int

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

tau_th

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

v_thresh

spynnaker.pyNN.models.neuron.threshold_types.threshold_type_static module

class spynnaker.pyNN.models.neuron.threshold_types.threshold_type_static.**ThresholdTypeStat**
Bases: *spynnaker.pyNN.models.neuron.threshold_types.abstract_threshold_type.AbstractThresholdType*

A threshold that is a static value

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the state variables

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (`int`) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

v_thresh

Module contents

class spynnaker.pyNN.models.neuron.threshold_types.**AbstractThresholdType** (*data_types*)
Bases: *spynnaker.pyNN.models.neuron.implementations.abstract_standard_neuron_component.AbstractStandardNeuronComponent*

Represents types of threshold for a neuron (e.g., stochastic).

Parameters **data_types** – A list of data types in the component structure, in the order that they appear

class spynnaker.pyNN.models.neuron.threshold_types.**ThresholdTypeStatic** (*v_thresh*)
Bases: *spynnaker.pyNN.models.neuron.threshold_types.abstract_threshold_type.AbstractThresholdType*

A threshold that is a static value

add_parameters(*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the parameters

add_state_variables(*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the state variables

get_n_cpu_cycles(*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (`int`) – The number of neurons to get the cycles for

Return type int

get_units(*variable*)
Get the units of the given variable

Parameters **variable** (`str`) – The name of the variable

get_values(*parameters*, *state_variables*, *vertex_slice*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable(*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (`str`) – The name of the variable

Return type bool

update_values(*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

v_thresh

class `spynnaker.pyNN.models.neuron.threshold_types.`**ThresholdTypeMaassStochastic**(*du_th*, *tau_th*, *v_thresh*)
Bases: *`spynnaker.pyNN.models.neuron.threshold_types.abstract_threshold_type.AbstractThresholdType`*

A stochastic threshold

add_parameters (*parameters*)
Add the initial values of the parameters to the parameter holder

Parameters **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the parameters

add_state_variables (*state_variables*)
Add the initial values of the state variables to the state variables holder

Parameters **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – A holder of the state variables

du_th

get_n_cpu_cycles (*n_neurons*)
Get the number of CPU cycles required to update the state

Parameters **n_neurons** (*int*) – The number of neurons to get the cycles for

Return type int

get_units (*variable*)
Get the units of the given variable

Parameters **variable** (*str*) – The name of the variable

get_values (*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the parameters
- **state_variables** (`spinn_utilities.ranged.range_dictionary.RangeDictionary`) – The holder of the state variables
- **vertex_slice** – The slice of variables being retrieved

Returns A list with the same length as self.struct.field_types

Return type A list of (single value or list of values or RangedList)

has_variable (*variable*)
Determine if this component has a variable by the given name

Parameters **variable** (*str*) – The name of the variable

Return type bool

tau_th

update_values (*values*, *parameters*, *state_variables*)
Update the parameters and state variables with the given struct values that have been read from the machine

Parameters

- **values** – The values read from the machine, one for each struct element
- **parameters** – The holder of the parameters to update
- **state_variables** – The holder of the state variables to update

v_thresh

Submodules

spynnaker.pyNN.models.neuron.abstract_population_vertex module

class spynnaker.pyNN.models.neuron.abstract_population_vertex.**AbstractPopulationVertex**(*n_ne... label, constrai... max_... spike... ring_... incoming_s... neuron_... pynn...*)

Bases: pacman.model.graphs.application.application_vertex.ApplicationVertex, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification, spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, *spynnaker.pyNN.models.abstract_models.abstract_contains_units.AbstractContainsUnits*, *spynnaker.pyNN.models.common.abstract_spike_recordable.AbstractSpikeRecordable*, *spynnaker.pyNN.models.common.abstract_neuron_recordable.AbstractNeuronRecordable*, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.AbstractProvidesOutgoingPartitionCons... spinn_front_end_common.abstract_models.abstract_provides_incoming_partition_constraint... AbstractProvidesIncomingPartitionConstraints, *spynnaker.pyNN.models.abstract_models.abstract_population_initializable.AbstractPopulationInitializable*, *spynnaker.pyNN.models.abstract_models.abstract_population_settable.AbstractPopulationSettable*, spinn_front_end_common.abstract_models.abstract_changable_after_run.AbstractChangableAfterRun, spinn_front_end_common.abstract_models.abstract_rewrites_data_specification.AbstractRewritesDataSpecification, *spynnaker.pyNN.models.abstract_models.abstract_read_parameters_before_set.AbstractReadParametersBeforeSet*, *spynnaker.pyNN.models.abstract_models.abstract_accepts_incoming_synapses.AbstractAcceptsIncomingSynapses*, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl, spinn_front_end_common.abstract_models.abstract_can_reset.AbstractCanReset

Underlying vertex model for Neural Populations.

BASIC_MALLOC_USAGE = 2

BYTES_TILL_START_OF_GLOBAL_PARAMETERS = 32

RUNTIME_SDP_PORT_SIZE = 4

SPIKE_RECORDING_REGION = 0

TRAFFIC_IDENTIFIER = 'BufferTraffic'

add_pre_run_connection_holder(*connection_holder, edge, synapse_info*)
Add a connection holder to the vertex to be filled in when the connections are actually generated.

clear_connection_cache()
Clear the connection data stored in the vertex so far.

clear_recording(*variable*, *buffer_manager*, *placements*, *graph_mapper*)
Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

clear_spike_recording(*buffer_manager*, *placements*, *graph_mapper*)
Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

conductance_based

create_machine_vertex(*vertex_slice*, *resources_required*, *label=None*, *constraints=None*)
Create a machine vertex from this application vertex

Parameters

- **vertex_slice** (*Slice*) – The slice of atoms that the machine vertex will cover
- **resources_required** (*ResourceContainer*) – the resources used by the machine vertex
- **label** (*str or None*) – human readable label for the machine vertex
- **constraints** (*iterable(AbstractConstraint)*) – Constraints to be passed on to the machine vertex

describe()
Get a human-readable description of the cell or synapse type.

The output may be customised by specifying a different template together with an associated template engine (see `pyNN.descriptions`).

If template is None, then a dictionary containing the template context will be returned.

gen_on_machine(*vertex_slice*)

generate_data_specification(*spec*, *placement*, *machine_time_step*, *time_scale_factor*, *graph_mapper*, *application_graph*, *machine_graph*, *routing_info*, *data_n_time_steps*)
Generate a data specification.

Parameters

- **spec** (*DataSpecificationGenerator*) – The data specification to write to
- **placement** (*Placement*) – the placement the vertex is located at

Return type None

get_binary_file_name()
Get the binary name to be run for this vertex.

Return type str

get_binary_start_type()
Get the start type of the binary to be run.

Return type ExecutableType

get_connection_holders()

get_connections_from_machine(*transceiver*, *placement*, *edge*, *graph_mapper*, *routing_infos*, *synapse_information*, *machine_time_step*, *using_extra_monitor_cores*, *placements=None*, *monitor_api=None*, *monitor_placement=None*, *monitor_cores=None*, *handle_time_out_configuration=True*, *fixed_routes=None*)
Get the connections from the machine post-run.

get_cpu_usage_for_atoms(*vertex_slice*)

get_data(*variable*, *n_machine_time_steps*, *placements*, *graph_mapper*, *buffer_manager*, *machine_time_step*)
Get the recorded data

Parameters

- **variable** –
- **n_machine_time_steps** –
- **placements** –
- **graph_mapper** –
- **buffer_manager** –
- **machine_time_step** –

Returns

get_dtcm_usage_for_atoms(*vertex_slice*)

get_incoming_partition_constraints(*partition*)
Get constraints to be added to the given edge that goes in to a vertex of this vertex.

Parameters

- **partition** (*AbstractOutgoingEdgePartition*) – An partition that goes in to this vertex
- **partition** – partition that goes into this vertex

Returns A list of constraints

Return type list(AbstractConstraint) Gets the constraints for partitions going into this vertex.

Returns list of constraints

get_initial_value(*variable*, *selector=None*)
Gets the value for any variable whose in initialize_parameters.keys

Should return the current value not the default one.

Must support the variable as listed in initialize_parameters.keys, ideally also with *_init* removed or added.

Parameters

- **variable** (*str*) – variable name with our without _init
- **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

Returns A list or an Object which act like a list

get_maximum_delay_supported_in_ms (*machine_time_step*)
Get the maximum delay supported by this vertex.

get_neuron_sampling_interval (*variable*)
Returns the current sampling interval for this variable

Parameters **variable** – PyNN name of the variable

Returns Sampling interval in micro seconds

get_outgoing_partition_constraints (*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters

- **partition** (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex
- **partition** – the partition that leaves this vertex

Returns A list of constraints

Return type list(AbstractConstraint) Gets the constraints for partitions going out of this vertex.

Returns list of constraints

get_recordable_variables ()
Returns a list of the variables this models is expected to collect

get_resources_used_by_atoms (*vertex_slice, graph, machine_time_step*)
Get the separate resource requirements for a range of atoms

Parameters **vertex_slice** (*Slice*) – the low value of atoms to calculate resources from

Returns a Resource container that contains a CPUCyclesPerTickResource, DTCMResource and SDRAMResource

Return type ResourceContainer

Raises **None** – this method does not raise any known exception

get_spikes (*placements, graph_mapper, buffer_manager, machine_time_step*)
Get the recorded spikes from the object

Parameters

- **placements** – the placements object
- **graph_mapper** – the graph mapper object
- **buffer_manager** – the buffer manager object
- **machine_time_step** – the time step of the simulation

Returns A numpy array of 2-element arrays of (neuron_id, time) ordered by time

get_spikes_sampling_interval ()
Return the current sampling interval for spikes

Returns Sampling interval in micro seconds

get_synapse_id_by_target (*target*)
Get the ID of a synapse given the name.

Parameters **target** (*str*) – The name of the synapse

Return type int

get_units (*variable*)
Get units for a given variable

Parameters **variable** – the variable to find units from

Returns the units as a string.

get_value (*key*)
Get a property Get a property of the overall model.

initialize (*variable*, *value*)
Set the initial value of one of the state variables of the neurons in this population.

initialize_parameters
List the parameters that are initializable.

If "foo" is initializable there should be a setter initialize_foo and a getter property foo_init

Returns list of property names

is_recording (*variable*)
Determines if variable is being recorded

Returns True if variable are being recorded, False otherwise

Return type bool

is_recording_spikes ()
Determine if spikes are being recorded

Returns True if spikes are being recorded, False otherwise

Return type bool

mark_no_changes ()
Marks the point after which changes are reported, so that new changes can be detected before the next check.

mark_regions_reloaded ()
Indicate that the regions have been reloaded

n_atoms
The number of atoms in the vertex

Return type int

read_parameters_from_machine (*transceiver*, *placement*, *vertex_slice*)
Read the parameters from the machine before any are changed

Parameters

- **transceiver** – the SpinnMan interface
- **placement** – the placement of a vertex
- **vertex_slice** – the slice of atoms for this vertex

regenerate_data_specification (*spec*, *placement*, *machine_time_step*, *time_scale_factor*, *graph_mapper*, *routing_info*)
Regenerate the data specification, only generating regions that have changed and need to be reloaded

Parameters

- **spec** (*DataSpecificationGenerator*) – Where to write the regenerated spec
- **placement** (*Placement*) – Where are we regenerating for?

requires_data_generation
True if changes that have been made require that data generation be performed. By default this returns False but can be overridden to indicate changes that require data regeneration.

Return type bool

requires_mapping
True if changes that have been made require that mapping be performed. By default this returns False but can be overridden to indicate changes that require mapping.

Return type bool

requires_memory_regions_to_be_reloaded()
Return true if any data region needs to be reloaded

Return type bool

reset_to_first_timestep()
Reset the object to first time step.

ring_buffer_sigma

set_initial_value(*variable*, *value*, *selector=None*)
Sets the value for any variable whose in initialize_parameters.keys

Must support the variable as listed in initialize_parameters.keys, ideally also with *_init* removed or added

Parameters

- **variable** (*str*) – variable name with our without *_init*
- **value** – New value for the variable
- **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

Returns A list or an Object which act like a list

set_recording(*variable*, *new_state=True*, *sampling_interval=None*, *indexes=None*)
Sets variable to being recorded

set_recording_spikes(*new_state=True*, *sampling_interval=None*, *indexes=None*)
Set spikes to being recorded. If new_state is false all other parameters are ignored.

Parameters

- **new_state** (*bool*) – Set if the spikes are recording or not
- **sampling_interval** – The interval at which spikes are recorded. Must be a whole multiple of the timestep None will be taken as the timestep
- **indexes** – The indexes of the neurons that will record spikes. If None the assumption is all neurons are recording

set_synapse_dynamics(*synapse_dynamics*)
Set the synapse dynamics of this vertex.

set_value(*key*, *value*)
Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign Set a property of the overall model.

spikes_per_second

synapse_dynamics

weight_scale

spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model module

class spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model.**AbstractPyNNNeuronModel**(*model*

Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex(*n_neurons, label, constraints, spikes_per_second, ring_buffer_sigma, incoming_spike_buffer_size*)

Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {'incoming_spike_buffer_size': None, 'ring_buffer_sig

classmethod get_max_atoms_per_core()

Get the maximum number of atoms per core for this model

Return type int

classmethod set_model_max_atoms_per_core(*n_atoms=255*)

Set the maximum number of atoms per core for this model

Parameters n_atoms (*int or None*) – The new maximum, or None for the largest possible

spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard module

class spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.**AbstractPyNNNeuronMo**

Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model.AbstractPyNNNeuronModel*

spynnaker.pyNN.models.neuron.connection_holder module

class spynnaker.pyNN.models.neuron.connection_holder.**ConnectionHolder**(*data_items_to_return*, *as_list*, *n_pre_atoms*, *n_post_atoms*, *connections=None*, *fixed_values=None*, *notify=None*)

Bases: `object`

Holds a set of connections to be returned in a PyNN-specific format

Parameters

- **data_items_to_return** – A list of data fields to be returned
- **as_list** – True if the data will be returned as a list, False if it is to be returned as a matrix (or series of matrices)
- **n_pre_atoms** – The number of atoms in the pre-vertex
- **n_post_atoms** – The number of atoms in the post-vertex
- **connections** – Any initial connections, as a numpy structured array of source, target, weight and delay
- **fixed_values** – A list of tuples of field names and fixed values to be appended to the other fields per connection, formatted as [(field_name, value), . . .]. Note that if the field is to be returned, the name must also appear in data_items_to_return, which determines the order of items in the result
- **notify** – A callback to call when the connections have all been added. This should accept a single parameter, which will contain the data requested

add_connections(*connections*)

Add connections to the holder to be returned

Parameters connections – The connection to add, as a numpy structured array of source, target, weight and delay

connections

The connections stored

finish()

Finish adding connections

spynnaker.pyNN.models.neuron.generator_data module

class spynnaker.pyNN.models.neuron.generator_data.**GeneratorData**(*synaptic_matrix_offset*, *delayed_synaptic_matrix_offset*, *max_row_n_words*, *max_delayed_row_n_words*, *max_row_n_synapses*, *max_delayed_row_n_synapses*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_information*, *max_stage*, *machine_time_step*)

Bases: object

Data for each connection of the synapse generator.

BASE_SIZE = 68

gen_data

Get the data to be written for this connection

Return type numpy array of uint32

size

The size of the generated data in bytes

Return type int

spynnaker.pyNN.models.neuron.population_machine_vertex module

class spynnaker.pyNN.models.neuron.population_machine_vertex.**PopulationMachineVertex**(*resource recorded label*, *constraints*)

Bases: pacman.model.graphs.machine.machine_vertex.MachineVertex, spinn_front_end_common.interface.buffer_management.buffer_models.abstract_receive_buffers_to_host.AbstractReceiveBuffersToHost, spinn_front_end_common.interface.provenance.provides_provenance_data_from_machine_impl.ProvidesProvenanceDataFromMachineImpl, spinn_front_end_common.abstract_models.abstract_recordable.AbstractRecordable, spinn_front_end_common.interface.profiling.abstract_has_profile_data.AbstractHasProfileData

Parameters

- **resources_required** –
- **recorded_region_ids** –

- **label** –
- **constraints** –

class EXTRA_PROVENANCE_DATA_ENTRIES
Bases: `enum.Enum`

An enumeration.

BUFFER_OVERFLOW_COUNT = 2

CURRENT_TIMER_TIC = 3

PLASTIC_SYNAPTIC_WEIGHT_SATURATION_COUNT = 4

PRE_SYNAPTIC_EVENT_COUNT = 0

SATURATION_COUNT = 1

N_ADDITIONAL_PROVENANCE_DATA_ITEMS = 5

PROFILE_TAG_LABELS = {0: 'TIMER', 1: 'DMA_READ', 2: 'INCOMING_SPIKE', 3: 'PROCESS_I

get_profile_data(*transceiver*, *placement*)
Get the profile data recorded during simulation

Return type `spinn_front_end_common.interface.profiling.profile_data.ProfileData`

get_provenance_data_from_machine(*transceiver*, *placement*)
Retrieve the provenance data.

Parameters

- **transceiver** (*Transceiver*) – How to talk to the machine
- **placement** (*Placement*) – Which vertex are we retrieving from, and where was it

Return type list(ProvenanceDataItem)

get_recorded_region_ids()
Get the recording region IDs that have been recorded using buffering

Returns The region numbers that have active recording

Return type iterable(int)

get_recording_region_base_address(*txrx*, *placement*)
Get the recording region base address

Parameters

- **txrx** (*Transceiver*) – the SpiNNMan instance
- **placement** (*Placement*) – the placement object of the core to find the address of

Returns the base address of the recording region

Return type int

is_recording()
Deduce if the recorder is actually recording

Return type bool

resources_required
The resources required by the vertex

Return type ResourceContainer

spynnaker.pyNN.models.neuron.synaptic_manager module

class spynnaker.pyNN.models.neuron.synaptic_manager.**SynapticManager**(*n_synapse_types*, *ring_buffer_sigma*, *spikes_per_second*, *config*, *population_table_type=None*, *synapse_io=None*)

Bases: object

Deals with synapses

add_pre_run_connection_holder(*connection_holder*, *edge*, *synapse_info*)

clear_connection_cache()

gen_on_machine(*vertex_slice*)
True if the synapses should be generated on the machine

get_connection_holders()

get_connections_from_machine(*transceiver*, *placement*, *machine_edge*, *graph_mapper*, *routing_infos*, *synapse_info*, *machine_time_step*, *using_extra_monitor_cores*, *placements=None*, *monitor_api=None*, *monitor_placement=None*, *monitor_cores=None*, *handle_time_out_configuration=True*, *fixed_routes=None*)

get_dtcm_usage_in_bytes()

get_incoming_partition_constraints()

get_maximum_delay_supported_in_ms(*machine_time_step*)

get_n_cpu_cycles()

get_sdram_usage_in_bytes(*vertex_slice*, *in_edges*, *machine_time_step*)

ring_buffer_sigma

spikes_per_second

synapse_dynamics

vertex_executable_suffix

write_data_spec(*spec*, *application_vertex*, *post_vertex_slice*, *machine_vertex*, *placement*, *machine_graph*, *application_graph*, *routing_info*, *graph_mapper*, *weight_scale*, *machine_time_step*)

Module contents

class spynnaker.pyNN.models.neuron.**AbstractPopulationVertex**(*n_neurons*, *label*, *constraints*, *max_atoms_per_core*, *spikes_per_second*, *ring_buffer_sigma*, *incoming_spike_buffer_size*, *neuron_impl*, *pynn_model*)

Bases: pacman.model.graphs.application.application_vertex.ApplicationVertex, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification, spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, *spynnaker.pyNN.models.abstract_models.abstract_contains_units.AbstractContainsUnits*, *spynnaker.pyNN.models.common.abstract_spike_recordable.AbstractSpikeRecordable*, *spynnaker.pyNN.models.common.abstract_neuron_recordable.AbstractNeuronRecordable*, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.AbstractProvidesOutgoingPartitionConst spinn_front_end_common.abstract_models.abstract_provides_incoming_partition_constraint AbstractProvidesIncomingPartitionConstraints, *spynnaker.pyNN.models.abstract_models.abstract_population_initializable.AbstractPopulationInitializable*, *spynnaker.pyNN.models.abstract_models.abstract_population_settable.AbstractPopulationSettable*, spinn_front_end_common.abstract_models.abstract_changable_after_run.AbstractChangableAfterRun, spinn_front_end_common.abstract_models.abstract_rewrites_data_specification.AbstractRewritesDataSpecification, *spynnaker.pyNN.models.abstract_models.abstract_read_parameters_before_set.AbstractReadParametersBeforeSet*, *spynnaker.pyNN.models.abstract_models.abstract_accepts_incoming_synapses.AbstractAcceptsIncomingSynapses*, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl, spinn_front_end_common.abstract_models.abstract_can_reset.AbstractCanReset

Underlying vertex model for Neural Populations.

BASIC_MALLOC_USAGE = 2

BYTES_TILL_START_OF_GLOBAL_PARAMETERS = 32

RUNTIME_SDP_PORT_SIZE = 4

SPIKE_RECORDING_REGION = 0

TRAFFIC_IDENTIFIER = 'BufferTraffic'

add_pre_run_connection_holder(*connection_holder*, *edge*, *synapse_info*)
Add a connection holder to the vertex to be filled in when the connections are actually generated.

clear_connection_cache()
Clear the connection data stored in the vertex so far.

clear_recording(*variable*, *buffer_manager*, *placements*, *graph_mapper*)
Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

clear_spike_recording(*buffer_manager*, *placements*, *graph_mapper*)
Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

conductance_based

create_machine_vertex(*vertex_slice*, *resources_required*, *label=None*, *constraints=None*)
Create a machine vertex from this application vertex

Parameters

- **vertex_slice** (`Slice`) – The slice of atoms that the machine vertex will cover
- **resources_required** (`ResourceContainer`) – the resources used by the machine vertex
- **label** (`str or None`) – human readable label for the machine vertex
- **constraints** (`iterable(AbstractConstraint)`) – Constraints to be passed on to the machine vertex

describe()
Get a human-readable description of the cell or synapse type.

The output may be customised by specifying a different template together with an associated template engine (see `pyNN.descriptions`).

If template is None, then a dictionary containing the template context will be returned.

gen_on_machine(*vertex_slice*)

generate_data_specification(*spec*, *placement*, *machine_time_step*, *time_scale_factor*, *graph_mapper*, *application_graph*, *machine_graph*, *routing_info*, *data_n_time_steps*)
Generate a data specification.

Parameters

- **spec** (`DataSpecificationGenerator`) – The data specification to write to
- **placement** (`Placement`) – the placement the vertex is located at

Return type None

get_binary_file_name()
Get the binary name to be run for this vertex.

Return type str

get_binary_start_type()
Get the start type of the binary to be run.

Return type ExecutableType

get_connection_holders()

get_connections_from_machine(*transceiver, placement, edge, graph_mapper, routing_infos, synapse_information, machine_time_step, using_extra_monitor_cores, placements=None, monitor_api=None, monitor_placement=None, monitor_cores=None, handle_time_out_configuration=True, fixed_routes=None*)
Get the connections from the machine post-run.

get_cpu_usage_for_atoms(*vertex_slice*)

get_data(*variable, n_machine_time_steps, placements, graph_mapper, buffer_manager, machine_time_step*)
Get the recorded data

Parameters

- **variable** –
- **n_machine_time_steps** –
- **placements** –
- **graph_mapper** –
- **buffer_manager** –
- **machine_time_step** –

Returns

get_dtcm_usage_for_atoms(*vertex_slice*)

get_incoming_partition_constraints(*partition*)
Get constraints to be added to the given edge that goes in to a vertex of this vertex.

Parameters

- **partition** (*AbstractOutgoingEdgePartition*) – An partition that goes in to this vertex
- **partition** – partition that goes into this vertex

Returns A list of constraints

Return type list(AbstractConstraint) Gets the constraints for partitions going into this vertex.

Returns list of constraints

get_initial_value(*variable, selector=None*)
Gets the value for any variable whose in initialize_parameters.keys

Should return the current value not the default one.

Must support the variable as listed in initialize_parameters.keys, ideally also with *_init* removed or added.

Parameters

- **variable** (*str*) – variable name with our without _init
- **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

Returns A list or an Object which act like a list

get_maximum_delay_supported_in_ms(*machine_time_step*)
Get the maximum delay supported by this vertex.

get_neuron_sampling_interval(*variable*)
Returns the current sampling interval for this variable

Parameters **variable** – PyNN name of the variable

Returns Sampling interval in micro seconds

get_outgoing_partition_constraints(*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters

- **partition** (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex
- **partition** – the partition that leaves this vertex

Returns A list of constraints

Return type list(AbstractConstraint) Gets the constraints for partitions going out of this vertex.

Returns list of constraints

get_recordable_variables()
Returns a list of the variables this models is expected to collect

get_resources_used_by_atoms(*vertex_slice*, *graph*, *machine_time_step*)
Get the separate resource requirements for a range of atoms

Parameters **vertex_slice** (*Slice*) – the low value of atoms to calculate resources from

Returns a Resource container that contains a CPUCyclesPerTickResource, DTCMResource and SDRAMResource

Return type ResourceContainer

Raises **None** – this method does not raise any known exception

get_spikes(*placements*, *graph_mapper*, *buffer_manager*, *machine_time_step*)
Get the recorded spikes from the object

Parameters

- **placements** – the placements object
- **graph_mapper** – the graph mapper object
- **buffer_manager** – the buffer manager object
- **machine_time_step** – the time step of the simulation

Returns A numpy array of 2-element arrays of (neuron_id, time) ordered by time

get_spikes_sampling_interval()
Return the current sampling interval for spikes

Returns Sampling interval in micro seconds

get_synapse_id_by_target(*target*)
Get the ID of a synapse given the name.

Parameters **target** (*str*) – The name of the synapse

Return type int

get_units(*variable*)
Get units for a given variable

Parameters **variable** – the variable to find units from

Returns the units as a string.

get_value(*key*)
Get a property Get a property of the overall model.

initialize(*variable*, *value*)
Set the initial value of one of the state variables of the neurons in this population.

initialize_parameters
List the parameters that are initializable.

If "foo" is initializable there should be a setter initialize_foo and a getter property foo_init

Returns list of property names

is_recording(*variable*)
Determines if variable is being recorded

Returns True if variable are being recorded, False otherwise

Return type bool

is_recording_spikes()
Determine if spikes are being recorded

Returns True if spikes are being recorded, False otherwise

Return type bool

mark_no_changes()
Marks the point after which changes are reported, so that new changes can be detected before the next check.

mark_regions_reloaded()
Indicate that the regions have been reloaded

n_atoms
The number of atoms in the vertex

Return type int

read_parameters_from_machine(*transceiver*, *placement*, *vertex_slice*)
Read the parameters from the machine before any are changed

Parameters

- **transceiver** – the SpinnMan interface
- **placement** – the placement of a vertex
- **vertex_slice** – the slice of atoms for this vertex

regenerate_data_specification(*spec*, *placement*, *machine_time_step*, *time_scale_factor*, *graph_mapper*, *routing_info*)
Regenerate the data specification, only generating regions that have changed and need to be reloaded

Parameters

- **spec** (`DataSpecificationGenerator`) – Where to write the regenerated spec
- **placement** (`Placement`) – Where are we regenerating for?

requires_data_generation
True if changes that have been made require that data generation be performed. By default this returns False but can be overridden to indicate changes that require data regeneration.

Return type bool

requires_mapping
True if changes that have been made require that mapping be performed. By default this returns False but can be overridden to indicate changes that require mapping.

Return type bool

requires_memory_regions_to_be_reloaded()
Return true if any data region needs to be reloaded

Return type bool

reset_to_first_timestep()
Reset the object to first time step.

ring_buffer_sigma

set_initial_value(*variable*, *value*, *selector=None*)
Sets the value for any variable whose in initialize_parameters.keys

Must support the variable as listed in initialize_parameters.keys, ideally also with *_init* removed or added

Parameters

- **variable** (*str*) – variable name with our without *_init*
- **value** – New value for the variable
- **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

Returns A list or an Object which act like a list

set_recording(*variable*, *new_state=True*, *sampling_interval=None*, *indexes=None*)
Sets variable to being recorded

set_recording_spikes(*new_state=True*, *sampling_interval=None*, *indexes=None*)
Set spikes to being recorded. If new_state is false all other parameters are ignored.

Parameters

- **new_state** (*bool*) – Set if the spikes are recording or not
- **sampling_interval** – The interval at which spikes are recorded. Must be a whole multiple of the timestep None will be taken as the timestep
- **indexes** – The indexes of the neurons that will record spikes. If None the assumption is all neurons are recording

set_synapse_dynamics(*synapse_dynamics*)
Set the synapse dynamics of this vertex.

set_value(*key*, *value*)
Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign Set a property of the overall model.

spikes_per_second

synapse_dynamics

weight_scale

class spynnaker.pyNN.models.neuron.**ConnectionHolder**(*data_items_to_return*, *as_list*, *n_pre_atoms*, *n_post_atoms*, *connections=None*, *fixed_values=None*, *notify=None*)

Bases: `object`

Holds a set of connections to be returned in a PyNN-specific format

Parameters

- **data_items_to_return** – A list of data fields to be returned
- **as_list** – True if the data will be returned as a list, False if it is to be returned as a matrix (or series of matrices)
- **n_pre_atoms** – The number of atoms in the pre-vertex
- **n_post_atoms** – The number of atoms in the post-vertex
- **connections** – Any initial connections, as a numpy structured array of source, target, weight and delay
- **fixed_values** – A list of tuples of field names and fixed values to be appended to the other fields per connection, formatted as [(field_name, value), . . .]. Note that if the field is to be returned, the name must also appear in data_items_to_return, which determines the order of items in the result
- **notify** – A callback to call when the connections have all been added. This should accept a single parameter, which will contain the data requested

add_connections(*connections*)

Add connections to the holder to be returned

Parameters connections – The connection to add, as a numpy structured array of source, target, weight and delay

connections

The connections stored

finish()

Finish adding connections

class spynnaker.pyNN.models.neuron.**SynapticManager**(*n_synapse_types*, *ring_buffer_sigma*, *spikes_per_second*, *config*, *population_table_type=None*, *synapse_io=None*)

Bases: `object`

Deals with synapses

add_pre_run_connection_holder(*connection_holder*, *edge*, *synapse_info*)

clear_connection_cache()

gen_on_machine(*vertex_slice*)

True if the synapses should be generated on the machine

get_connection_holders()

get_connections_from_machine(*transceiver, placement, machine_edge, graph_mapper, routing_infos, synapse_info, machine_time_step, using_extra_monitor_cores, placements=None, monitor_api=None, monitor_placement=None, monitor_cores=None, handle_time_out_configuration=True, fixed_routes=None*)

get_dtcm_usage_in_bytes()

get_incoming_partition_constraints()

get_maximum_delay_supported_in_ms(*machine_time_step*)

get_n_cpu_cycles()

get_sdram_usage_in_bytes(*vertex_slice, in_edges, machine_time_step*)

ring_buffer_sigma

spikes_per_second

synapse_dynamics

vertex_executable_suffix

write_data_spec(*spec, application_vertex, post_vertex_slice, machine_vertex, placement, machine_graph, application_graph, routing_info, graph_mapper, weight_scale, machine_time_step*)

class spynnaker.pyNN.models.neuron.**PopulationMachineVertex**(*resources_required, recorded_region_ids, label, constraints*)

Bases: pacman.model.graphs.machine.machine_vertex.MachineVertex, spinn_front_end_common.interface.buffer_management.buffer_models.abstract_receive_buffers_to_host.AbstractReceiveBuffersToHost, spinn_front_end_common.interface.provenance.provides_provenance_data_from_machine_impl.ProvidesProvenanceDataFromMachineImpl, spinn_front_end_common.abstract_models.abstract_recordable.AbstractRecordable, spinn_front_end_common.interface.profiling.abstract_has_profile_data.AbstractHasProfileData

Parameters

- **resources_required** –
- **recorded_region_ids** –
- **label** –
- **constraints** –

class EXTRA_PROVENANCE_DATA_ENTRIES

Bases: enum.Enum

An enumeration.

BUFFER_OVERFLOW_COUNT = 2

CURRENT_TIMER_TIC = 3

PLASTIC_SYNAPTIC_WEIGHT_SATURATION_COUNT = 4

PRE_SYNAPTIC_EVENT_COUNT = 0

SATURATION_COUNT = 1

N_ADDITIONAL_PROVENANCE_DATA_ITEMS = 5

PROFILE_TAG_LABELS = {0: 'TIMER', 1: 'DMA_READ', 2: 'INCOMING_SPIKE', 3: 'PROCESS_

get_profile_data(*transceiver*, *placement*)
Get the profile data recorded during simulation

Return type spinn_front_end_common.interface.profiling.profile_data.ProfileData

get_provenance_data_from_machine(*transceiver*, *placement*)
Retrieve the provenance data.

Parameters

- **transceiver** (*Transceiver*) – How to talk to the machine
- **placement** (*Placement*) – Which vertex are we retrieving from, and where was it

Return type list(ProvenanceDataItem)

get_recorded_region_ids()
Get the recording region IDs that have been recorded using buffering

Returns The region numbers that have active recording

Return type iterable(int)

get_recording_region_base_address(*txrx*, *placement*)
Get the recording region base address

Parameters

- **txrx** (*Transceiver*) – the SpiNNMan instance
- **placement** (*Placement*) – the placement object of the core to find the address of

Returns the base address of the recording region

Return type int

is_recording()
Deduce if the recorder is actually recording

Return type bool

resources_required
The resources required by the vertex

Return type ResourceContainer

class spynnaker.pyNN.models.neuron.**AbstractPyNNNeuronModel**(*model*)
Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex(*n_neurons*, *label*, *constraints*, *spikes_per_second*, *ring_buffer_sigma*, *incoming_spike_buffer_size*)
Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {'incoming_spike_buffer_size': None, 'ring_buffer_sig

classmethod get_max_atoms_per_core()
Get the maximum number of atoms per core for this model

Return type int

classmethod set_model_max_atoms_per_core(*n_atoms=255*)
Set the maximum number of atoms per core for this model

Parameters **n_atoms** (*int or None*) – The new maximum, or None for the largest possible

class spynnaker.pyNN.models.neuron.**AbstractPyNNNeuronModelStandard**(*model_name, binary, neuron_model, input_type, synapse_type, threshold_type, additional_input_type=None*)
Bases: *spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model.AbstractPyNNNeuronModel*

spynnaker.pyNN.models.spike_source package

Submodules

spynnaker.pyNN.models.spike_source.spike_source_array module

class spynnaker.pyNN.models.spike_source.spike_source_array.**SpikeSourceArray**(*spike_times=[]*)
Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex(*n_neurons, label, constraints*)
Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {}

spynnaker.pyNN.models.spike_source.spike_source_array_vertex module

class spynnaker.pyNN.models.spike_source.spike_source_array_vertex.**SpikeSourceArrayVertex**(

Bases: spinn_front_end_common.utility_models.reverse_ip_tag_multi_cast_source.ReverseIpTagMultiCastSource, *spynnaker.pyNN.models.common.abstract_spike_recordable.AbstractSpikeRecordable*, *spynnaker.pyNN.models.common.simple_population_settable.SimplePopulationSettable*, spinn_front_end_common.abstract_models.abstract_changable_after_run.AbstractChangableAfterRun, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

Model for play back of spikes

SPIKE_RECORDING_REGION_ID = 0

clear_spike_recording(*buffer_manager*, *placements*, *graph_mapper*)

Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

describe()

Returns a human-readable description of the cell or synapse type.

The output may be customised by specifying a different template together with an associated template engine (see pyNN.descriptions).

If template is None, then a dictionary containing the template context will be returned.

get_spikes(*placements*, *graph_mapper*, *buffer_manager*, *machine_time_step*)

Get the recorded spikes from the object

Parameters

- **placements** – the placements object
- **graph_mapper** – the graph mapper object
- **buffer_manager** – the buffer manager object
- **machine_time_step** – the time step of the simulation

Returns A numpy array of 2-element arrays of (neuron_id, time) ordered by time

get_spikes_sampling_interval()

Return the current sampling interval for spikes

Returns Sampling interval in micro seconds

is_recording_spikes()
Determine if spikes are being recorded

Returns True if spikes are being recorded, False otherwise

Return type bool

mark_no_changes()
Marks the point after which changes are reported, so that new changes can be detected before the next check.

requires_mapping
True if changes that have been made require that mapping be performed. By default this returns False but can be overridden to indicate changes that require mapping.

Return type bool

set_recording_spikes(*new_state=True*, *sampling_interval=None*, *indexes=None*)
Set spikes to being recorded. If new_state is false all other parameters are ignored.

Parameters

- **new_state** (*bool*) – Set if the spikes are recording or not
- **sampling_interval** – The interval at which spikes are recorded. Must be a whole multiple of the timestep None will be taken as the timestep
- **indexes** – The indexes of the neurons that will record spikes. If None the assumption is all neurons are recording

spike_times
The spike times of the spike source array

spynnaker.pyNN.models.spike_source.spike_source_from_file module

class spynnaker.pyNN.models.spike_source.spike_source_from_file.**SpikeSourceFromFile**(*spike_tim min_atom max_atom min_time max_time split_valu*

Bases: *spynnaker.pyNN.models.spike_source.spike_source_array.SpikeSourceArray*

SpikeSourceArray that works from a file

spike_times

spynnaker.pyNN.models.spike_source.spike_source_poisson module

class spynnaker.pyNN.models.spike_source.spike_source_poisson.**SpikeSourcePoisson**(*rate=1.0*, *start=0*, *duration=None*)

Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex(*n_neurons*, *label*, *constraints*, *seed*, *max_rate*)
Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {'max_rate': None, 'seed': None}

classmethod get_max_atoms_per_core()
Get the maximum number of atoms per core for this model

Return type int

classmethod set_model_max_atoms_per_core(*n_atoms=500*)
Set the maximum number of atoms per core for this model

Parameters **n_atoms** (*int or None*) – The new maximum, or None for the largest possible

spynnaker.pyNN.models.spike_source.spike_source_poisson_machine_vertex module

class spynnaker.pyNN.models.spike_source.spike_source_poisson_machine_vertex.**SpikeSourcePo**

Bases: pacman.model.graphs.machine.machine_vertex.MachineVertex, spinn_front_end_common.interface.buffer_management.buffer_models.abstract_receive_buffers_to_host.AbstractReceiveBuffersToHost, spinn_front_end_common.interface.provenance.provides_provenance_data_from_machine_impl ProvidesProvenanceDataFromMachineImpl, spinn_front_end_common.abstract_models.abstract_recordable.AbstractRecordable, spinn_front_end_common.abstract_models.abstract_supports_database_injection.AbstractSupportsDatabaseInjection, spinn_front_end_common.interface.profiling.abstract_has_profile_data.AbstractHasProfileData

class POISSON_SPIKE_SOURCE_REGIONS
Bases: enum.Enum

An enumeration.

POISSON_PARAMS_REGION = 1

PROFILER_REGION = 4

PROVENANCE_REGION = 3

SPIKE_HISTORY_REGION = 2

SYSTEM_REGION = 0

PROFILE_TAG_LABELS = {0: 'TIMER', 1: 'PROB_FUNC'}

get_profile_data(*transceiver, placement*)
Get the profile data recorded during simulation

Return type spinn_front_end_common.interface.profiling.profile_data.ProfileData

get_recorded_region_ids()

Get the recording region IDs that have been recorded using buffering

Returns The region numbers that have active recording

Return type iterable(int)

get_recording_region_base_address(*txrx*, *placement*)

Get the recording region base address

Parameters

- **txrx** (*Transceiver*) – the SpiNNMan instance
- **placement** (*Placement*) – the placement object of the core to find the address of

Returns the base address of the recording region

Return type int

is_in_injection_mode(*graph*)

Whether this vertex is actually in injection mode.

Return type bool

is_recording()

Deduce if the recorder is actually recording

Return type bool

resources_required

The resources required by the vertex

Return type ResourceContainer

spynnaker.pyNN.models.spike_source.spike_source_poisson_vertex module

class spynnaker.pyNN.models.spike_source.spike_source_poisson_vertex.**SpikeSourcePoissonVert**

Bases: pacman.model.graphs.application.application_vertex.ApplicationVertex, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification, spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, *spynnaker.pyNN.models.common.abstract_spike_recordable.AbstractSpikeRecordable*, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.

AbstractProvidesOutgoingPartitionConstraints, spinn_front_end_common.abstract_models.abstract_changable_after_run.AbstractChangableAfterRun, *spynnaker.pyNN.models.abstract_models.abstract_read_parameters_before_set.AbstractReadParametersBeforeSet*, spinn_front_end_common.abstract_models.abstract_rewrites_data_specification.AbstractRewritesDataSpecification, *spynnaker.pyNN.models.common.simple_population_settable.SimplePopulationSettable*, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

A Poisson Spike source object

SPIKE_RECORDING_REGION_ID = 0

clear_spike_recording(*buffer_manager*, *placements*, *graph_mapper*)

Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

convert_rate(*rate*)

create_machine_vertex(*vertex_slice*, *resources_required*, *label=None*, *constraints=None*)

Create a machine vertex from this application vertex

Parameters

- **vertex_slice** (*Slice*) – The slice of atoms that the machine vertex will cover
- **resources_required** (*ResourceContainer*) – the resources used by the machine vertex
- **label** (*str or None*) – human readable label for the machine vertex
- **constraints** (*iterable(AbstractConstraint)*) – Constraints to be passed on to the machine vertex

describe()

Returns a human-readable description of the cell or synapse type.

The output may be customised by specifying a different template together with an associated template engine (see pyNN.descriptions).

If template is None, then a dictionary containing the template context will be returned.

duration

generate_data_specification(*spec*, *placement*, *machine_time_step*, *time_scale_factor*, *graph_mapper*, *routing_info*, *data_n_time_steps*, *graph*)

Generate a data specification.

Parameters

- **spec** (*DataSpecificationGenerator*) – The data specification to write to
- **placement** (*Placement*) – the placement the vertex is located at

Return type None

get_binary_file_name()

Get the binary name to be run for this vertex.

Return type str

get_binary_start_type()
Get the start type of the binary to be run.

Return type ExecutableType

static get_cpu_usage_for_atoms()

static get_dtcm_usage_for_atoms()

get_outgoing_partition_constraints(*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters **partition** (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

static get_params_bytes(*vertex_slice*)
Gets the size of the poisson parameters in bytes

Parameters **vertex_slice** –

get_recording_sdram_usage(*vertex_slice*, *machine_time_step*)

get_resources_used_by_atoms(*vertex_slice*, *machine_time_step*)
Get the separate resource requirements for a range of atoms

Parameters **vertex_slice** (*Slice*) – the low value of atoms to calculate resources from

Returns a Resource container that contains a CPUCyclesPerTickResource, DTCMResource and SDRAMResource

Return type ResourceContainer

Raises **None** – this method does not raise any known exception

get_spikes(*placements*, *graph_mapper*, *buffer_manager*, *machine_time_step*)
Get the recorded spikes from the object

Parameters

- **placements** – the placements object
- **graph_mapper** – the graph mapper object
- **buffer_manager** – the buffer manager object
- **machine_time_step** – the time step of the simulation

Returns A numpy array of 2-element arrays of (neuron_id, time) ordered by time

get_spikes_sampling_interval()
Return the current sampling interval for spikes

Returns Sampling interval in micro seconds

is_recording_spikes()
Determine if spikes are being recorded

Returns True if spikes are being recorded, False otherwise

Return type bool

mark_no_changes()
Marks the point after which changes are reported, so that new changes can be detected before the next check.

mark_regions_reloaded()
Indicate that the regions have been reloaded

max_rate

n_atoms
The number of atoms in the vertex

Return type int

rate

read_parameters_from_machine(*transceiver*, *placement*, *vertex_slice*)
Read the parameters from the machine before any are changed

Parameters

- **transceiver** – the SpinnMan interface
- **placement** – the placement of a vertex
- **vertex_slice** – the slice of atoms for this vertex

regenerate_data_specification(*spec*, *placement*, *machine_time_step*, *time_scale_factor*, *graph_mapper*, *routing_info*, *graph*)
Regenerate the data specification, only generating regions that have changed and need to be reloaded

Parameters

- **spec** (*DataSpecificationGenerator*) – Where to write the regenerated spec
- **placement** (*Placement*) – Where are we regenerating for?

requires_mapping
True if changes that have been made require that mapping be performed. By default this returns False but can be overridden to indicate changes that require mapping.

Return type bool

requires_memory_regions_to_be_reloaded()
Return true if any data region needs to be reloaded

Return type bool

reserve_memory_regions(*spec*, *placement*, *graph_mapper*)
Reserve memory regions for poisson source parameters and output buffer.

Parameters

- **spec** – the data specification writer
- **placement** – the location this vertex resides on in the machine
- **graph_mapper** – the mapping between app and machine graphs

Returns None

seed

set_recording_spikes(*new_state=True*, *sampling_interval=None*, *indexes=None*)
Set spikes to being recorded. If new_state is false all other parameters are ignored.

Parameters

- **new_state** (*bool*) – Set if the spikes are recording or not
- **sampling_interval** – The interval at which spikes are recorded. Must be a whole multiple of the timestep None will be taken as the timestep
- **indexes** – The indexes of the neurons that will record spikes. If None the assumption is all neurons are recording

set_value (*key*, *value*)
Set a property

Parameters

- **key** – the name of the parameter to change
- **value** – the new value of the parameter to assign

start

Module contents

class spynnaker.pyNN.models.spike_source.**SpikeSourceArray** (*spike_times=[]*)
Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex (*n_neurons*, *label*, *constraints*)
Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {}

class spynnaker.pyNN.models.spike_source.**SpikeSourceFromFile** (*spike_time_file*, *min_atom=None*, *max_atom=None*, *min_time=None*, *max_time=None*, *split_value='t'*)
Bases: *spynnaker.pyNN.models.spike_source.spike_source_array.SpikeSourceArray*

SpikeSourceArray that works from a file

spike_times

class spynnaker.pyNN.models.spike_source.**SpikeSourcePoisson** (*rate=1.0*, *start=0*, *duration=None*)
Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex (*n_neurons*, *label*, *constraints*, *seed*, *max_rate*)
Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population

- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {'max_rate': None, 'seed': None}

classmethod get_max_atoms_per_core()

Get the maximum number of atoms per core for this model

Return type int

classmethod set_model_max_atoms_per_core(*n_atoms=500*)

Set the maximum number of atoms per core for this model

Parameters n_atoms (*int or None*) – The new maximum, or None for the largest possible

spynnaker.pyNN.models.utility_models package

Subpackages

spynnaker.pyNN.models.utility_models.delays package

Submodules

spynnaker.pyNN.models.utility_models.delays.delay_block module

class spynnaker.pyNN.models.utility_models.delays.delay_block.**DelayBlock**(*n_delay_stages*, *delay_per_stage*, *vertex_slice*)

Bases: object

A block of delays for a vertex.

add_delay(*source_id*, *stage*)

delay_block

spynnaker.pyNN.models.utility_models.delays.delay_extension_machine_vertex module

class spynnaker.pyNN.models.utility_models.delays.delay_extension_machine_vertex.**DelayExten**

Bases: pacman.model.graphs.machine.machine_vertex.MachineVertex, spinn_front_end_common.interface.provenance.provides_provenance_data_from_machine_impl. ProvidesProvenanceDataFromMachineImpl

class EXTRA_PROVENANCE_DATA_ENTRIES

Bases: enum.Enum

An enumeration.

N_BUFFER_OVERFLOWS = 4

N_DELAYS = 5

N_PACKETS_ADDED = 2

N_PACKETS_PROCESSED = 1

N_PACKETS_RECEIVED = 0

N_PACKETS_SENT = 3

N_EXTRA_PROVENANCE_DATA_ENTRIES = 6

get_provenance_data_from_machine (*transceiver*, *placement*)
Retrieve the provenance data.

Parameters

- **transceiver** (*Transceiver*) – How to talk to the machine
- **placement** (*Placement*) – Which vertex are we retrieving from, and where was it

Return type list(ProvenanceDataItem)

resources_required
The resources required by the vertex

Return type ResourceContainer

spynnaker.pyNN.models.utility_models.delays.delay_extension_vertex module

class spynnaker.pyNN.models.utility_models.delays.delay_extension_vertex.**DelayExtensionVert**

Bases: pacman.model.graphs.application.application_vertex.ApplicationVertex, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification, spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.AbstractProvidesOutgoingPartitionCons spinn_front_end_common.abstract_models.abstract_provides_n_keys_for_partition.AbstractProvidesNKeysForPartition

Provide delays to incoming spikes in multiples of the maximum delays of a neuron (typically 16 or 32)

Parameters

- **n_neurons** – the number of neurons
- **delay_per_stage** – the delay per stage
- **source_vertex** – where messages are coming from
- **machine_time_step** – how long is the machine time step

- **timescale_factor** – what slowdown factor has been applied
- **constraints** – the vertex constraints
- **label** – the vertex label

add_delays (*vertex_slice*, *source_ids*, *stages*)
Add delayed connections for a given vertex slice

add_generator_data (*max_row_n_synapses*, *max_delayed_row_n_synapses*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_information*, *max_stage*, *machine_time_step*)
Add delays for a connection to be generated

create_machine_vertex (*vertex_slice*, *resources_required*, *label=None*, *constraints=None*)
Create a machine vertex from this application vertex

Parameters

- **vertex_slice** (`Slice`) – The slice of atoms that the machine vertex will cover
- **resources_required** (`ResourceContainer`) – the resources used by the machine vertex
- **label** (`str or None`) – human readable label for the machine vertex
- **constraints** (`iterable(AbstractConstraint)`) – Constraints to be passed on to the machine vertex

gen_on_machine (*vertex_slice*)
Determine if the given slice needs to be generated on the machine

generate_data_specification (*spec*, *placement*, *machine_graph*, *graph_mapper*, *routing_infos*)
Generate a data specification.

Parameters

- **spec** (`DataSpecificationGenerator`) – The data specification to write to
- **placement** (`Placement`) – the placement the vertex is located at

Return type None

get_binary_file_name ()
Get the binary name to be run for this vertex.

Return type str

get_binary_start_type ()
Get the start type of the binary to be run.

Return type ExecutableType

get_cpu_usage_for_atoms (*vertex_slice*)

get_dtcm_usage_for_atoms (*vertex_slice*)

get_n_keys_for_partition (*partition*, *graph_mapper*)
Get the number of keys required by the given partition of edges.

Parameters

- **partition** (`AbstractOutgoingEdgePartition`) – An partition that comes out of this vertex
- **graph_mapper** (`GraphMapper`) – A mapper between the graphs

Returns A list of constraints

Return type list(AbstractConstraint)

get_outgoing_partition_constraints(*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters **partition** (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

get_resources_used_by_atoms(*vertex_slice*, *graph*)
Get the separate resource requirements for a range of atoms

Parameters **vertex_slice** (*Slice*) – the low value of atoms to calculate resources from

Returns a Resource container that contains a CPUCyclesPerTickResource, DTCMResource and SDRAMResource

Return type ResourceContainer

Raises **None** – this method does not raise any known exception

get_sdram_usage_for_atoms(*out_edges*)

n_atoms
The number of atoms in the vertex

Return type int

n_delay_stages
The maximum number of delay stages required by any connection out of this delay extension vertex

source_vertex

write_delay_parameters(*spec*, *vertex_slice*, *key*, *incoming_key*, *incoming_mask*, *total_n_vertices*, *machine_time_step*, *time_scale_factor*, *n_outgoing_edges*)
Generate Delay Parameter data

write_setup_info(*spec*, *machine_time_step*, *time_scale_factor*)

spynnaker.pyNN.models.utility_models.delays.delay_generator_data module

class spynnaker.pyNN.models.utility_models.delays.delay_generator_data.**DelayGeneratorData**(

Bases: object

Data for each connection of the delay generator

BASE_SIZE = 32

gen_data
Get the data to be written for this connection

Return type numpy array of uint32

size
The size of the generated data in bytes

Return type int

Module contents

class spynnaker.pyNN.models.utility_models.delays.**DelayBlock**(*n_delay_stages*, *delay_per_stage*, *vertex_slice*)

Bases: object

A block of delays for a vertex.

add_delay(*source_id*, *stage*)

delay_block

class spynnaker.pyNN.models.utility_models.delays.**DelayExtensionMachineVertex**(*resources_required*, *label*, *constraints=None*)

Bases: pacman.model.graphs.machine.machine_vertex.MachineVertex, spinn_front_end_common.interface.provenance.provides_provenance_data_from_machine_impl.ProvidesProvenanceDataFromMachineImpl

class EXTRA_PROVENANCE_DATA_ENTRIES
Bases: enum.Enum

An enumeration.

N_BUFFER_OVERFLOWS = 4

N_DELAYS = 5

N_PACKETS_ADDED = 2

N_PACKETS_PROCESSED = 1

N_PACKETS_RECEIVED = 0

N_PACKETS_SENT = 3

N_EXTRA_PROVENANCE_DATA_ENTRIES = 6

get_provenance_data_from_machine(*transceiver*, *placement*)
Retrieve the provenance data.

Parameters

- **transceiver** (*Transceiver*) – How to talk to the machine
- **placement** (*Placement*) – Which vertex are we retrieving from, and where was it

Return type list(ProvenanceDataItem)

resources_required
The resources required by the vertex

Return type ResourceContainer

class spynnaker.pyNN.models.utility_models.delays.**DelayExtensionVertex**(*n_neurons*, *delay_per_stage*, *source_vertex*, *machine_time_step*, *timescale_factor*, *constraints=None*, *label='DelayExtension'*)

Bases: pacman.model.graphs.application.application_vertex.ApplicationVertex, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification, spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.AbstractProvidesOutgoingPartitionCons spinn_front_end_common.abstract_models.abstract_provides_n_keys_for_partition.AbstractProvidesNKeysForPartition

Provide delays to incoming spikes in multiples of the maximum delays of a neuron (typically 16 or 32)

Parameters

- **n_neurons** – the number of neurons
- **delay_per_stage** – the delay per stage
- **source_vertex** – where messages are coming from
- **machine_time_step** – how long is the machine time step
- **timescale_factor** – what slowdown factor has been applied
- **constraints** – the vertex constraints
- **label** – the vertex label

add_delays(*vertex_slice*, *source_ids*, *stages*)
Add delayed connections for a given vertex slice

add_generator_data(*max_row_n_synapses*, *max_delayed_row_n_synapses*, *pre_slices*, *pre_slice_index*, *post_slices*, *post_slice_index*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_information*, *max_stage*, *machine_time_step*)
Add delays for a connection to be generated

create_machine_vertex(*vertex_slice*, *resources_required*, *label=None*, *constraints=None*)
Create a machine vertex from this application vertex

Parameters

- **vertex_slice** (*Slice*) – The slice of atoms that the machine vertex will cover
- **resources_required** (*ResourceContainer*) – the resources used by the machine vertex
- **label** (*str or None*) – human readable label for the machine vertex

- **constraints** (*iterable(AbstractConstraint)*) – Constraints to be passed on to the machine vertex

gen_on_machine(*vertex_slice*)
Determine if the given slice needs to be generated on the machine

generate_data_specification(*spec, placement, machine_graph, graph_mapper, routing_infos*)
Generate a data specification.

Parameters

- **spec** (*DataSpecificationGenerator*) – The data specification to write to
- **placement** (*Placement*) – the placement the vertex is located at

Return type None

get_binary_file_name()
Get the binary name to be run for this vertex.

Return type str

get_binary_start_type()
Get the start type of the binary to be run.

Return type ExecutableType

get_cpu_usage_for_atoms(*vertex_slice*)

get_dtcm_usage_for_atoms(*vertex_slice*)

get_n_keys_for_partition(*partition, graph_mapper*)
Get the number of keys required by the given partition of edges.

Parameters

- **partition** (*AbstractOutgoingEdgePartition*) – An partition that comes out of this vertex
- **graph_mapper** (GraphMapper) – A mapper between the graphs

Returns A list of constraints

Return type list(AbstractConstraint)

get_outgoing_partition_constraints(*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters partition (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

get_resources_used_by_atoms(*vertex_slice, graph*)
Get the separate resource requirements for a range of atoms

Parameters vertex_slice (*Slice*) – the low value of atoms to calculate resources from

Returns a Resource container that contains a CPUCyclesPerTickResource, DTCMResource and SDRAMResource

Return type ResourceContainer

Raises None – this method does not raise any known exception

get_sdram_usage_for_atoms(*out_edges*)

n_atoms
The number of atoms in the vertex

Return type int

n_delay_stages
The maximum number of delay stages required by any connection out of this delay extension vertex

source_vertex

write_delay_parameters(*spec*, *vertex_slice*, *key*, *incoming_key*, *incoming_mask*, *total_n_vertices*, *machine_time_step*, *time_scale_factor*, *n_outgoing_edges*)
Generate Delay Parameter data

write_setup_info(*spec*, *machine_time_step*, *time_scale_factor*)

spynnaker.pyNN.models.utility_models.spike_injector package

Submodules

spynnaker.pyNN.models.utility_models.spike_injector.spike_injector module

class spynnaker.pyNN.models.utility_models.spike_injector.spike_injector.**SpikeInjector**
Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex(*n_neurons*, *label*, *constraints*, *port*, *virtual_key*, *reserve_reverse_ip_tag*)
Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {'port': None, 'reserve_reverse_ip_tag': False, 'vir

spynnaker.pyNN.models.utility_models.spike_injector.spike_injector_vertex module

class spynnaker.pyNN.models.utility_models.spike_injector.spike_injector_vertex.**SpikeInject**

Bases: spinn_front_end_common.utility_models.reverse_ip_tag_multi_cast_source.ReverseIpTagMultiCastSource, spinn_front_end_common.abstract_models.

abstract_provides_outgoing_partition_constraints.AbstractProvidesOutgoingPartitionCons spynnaker.pyNN.models.common.abstract_spike_recordable.AbstractSpikeRecordable, spynnaker.pyNN.models.common.simple_population_settable.SimplePopulationSettable

An Injector of Spikes for PyNN populations. This only allows the user to specify the virtual_key of the population to identify the population

SPIKE_RECORDING_REGION_ID = 0

clear_spike_recording(*buffer_manager*, *placements*, *graph_mapper*)
Clear the recorded data from the object

Parameters

- **buffer_manager** – the buffer manager object
- **placements** – the placements object
- **graph_mapper** – the graph mapper object

Return type None

default_parameters = {'label': 'spikeInjector', 'port': None, 'virtual_key': None}

describe()
Returns a human-readable description of the cell or synapse type.

The output may be customised by specifying a different template together with an associated template engine (see `pyNN.descriptions`).

If template is None, then a dictionary containing the template context will be returned.

get_outgoing_partition_constraints(*partition*)
Get constraints to be added to the given edge that comes out of this vertex.

Parameters partition (*AbstractOutgoingEdgePartition*) – An edge that comes out of this vertex

Returns A list of constraints

Return type list(AbstractConstraint)

get_spikes(*placements*, *graph_mapper*, *buffer_manager*, *machine_time_step*)
Get the recorded spikes from the object

Parameters

- **placements** – the placements object
- **graph_mapper** – the graph mapper object
- **buffer_manager** – the buffer manager object
- **machine_time_step** – the time step of the simulation

Returns A numpy array of 2-element arrays of (neuron_id, time) ordered by time

get_spikes_sampling_interval()
Return the current sampling interval for spikes

Returns Sampling interval in micro seconds

is_recording_spikes()
Determine if spikes are being recorded

Returns True if spikes are being recorded, False otherwise

Return type bool

port

set_recording_spikes (*new_state=True*, *sampling_interval=None*, *indexes=None*)
Set spikes to being recorded. If new_state is false all other parameters are ignored.

Parameters

- **new_state** (*bool*) – Set if the spikes are recording or not
- **sampling_interval** – The interval at which spikes are recorded. Must be a whole multiple of the timestep None will be taken as the timestep
- **indexes** – The indexes of the neurons that will record spikes. If None the assumption is all neurons are recording

virtual_key

Module contents

class spynnaker.pyNN.models.utility_models.spike_injector.**SpikeInjector**
Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex (*n_neurons*, *label*, *constraints*, *port*, *virtual_key*, *reserve_reverse_ip_tag*)
Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {'port': None, 'reserve_reverse_ip_tag': False, 'vir

spynnaker.pyNN.models.utility_models.synapse_expander package

Submodules

spynnaker.pyNN.models.utility_models.synapse_expander.synapse_expander module

spynnaker.pyNN.models.utility_models.synapse_expander.synapse_expander.**synapse_expander** (*app gra plac mem tran pro nan ex- e- cuta*

Run the synapse expander - needs to be done after data has been loaded

Module contents

Module contents

Submodules

spynnaker.pyNN.models.abstract_pynn_model module

class spynnaker.pyNN.models.abstract_pynn_model.**AbstractPyNNModel**
Bases: object

A Model that can be passed in to a Population object in PyNN

create_vertex(*n_neurons*, *label*, *constraints*)
Create a vertex for a population of the model

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label to give to the vertex
- **constraints** (*list or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_initial_values = {}

default_parameters = {}

default_population_parameters

Get the default values for the parameters at the population level These are parameters that can be passed in to the Population constructor in addition to the standard PyNN options

Return type dict(str, object)

classmethod get_max_atoms_per_core()
Get the maximum number of atoms per core for this model

Return type int

classmethod get_parameter_names()
Get the names of the parameters of the model

Return type list(str)

classmethod has_parameter(*name*)
Determine if the model has a parameter with the given name

Parameters name (*str*) – The name of the parameter to check for

Return type bool

classmethod set_model_max_atoms_per_core(*n_atoms=9223372036854775807*)
Set the maximum number of atoms per core for this model

Parameters n_atoms (*int or None*) – The new maximum, or None for the largest possible

spynnaker.pyNN.models.defaults module

spynnaker.pyNN.models.defaults.**default_initial_values**(*state_variables*)
Specifies arguments which are state variables. Only works on the __init__ method of a class that is additionally decorated with `defaults`()`

Parameters **state_variables** (`set of str`) – The names of the arguments that are state variables

spynnaker.pyNN.models.defaults.**default_parameters**(*parameters*)
Specifies arguments which are parameters. Only works on the __init__ method of a class that is additionally decorated with `defaults`()`

Parameters **parameters** (`set of str`) – The names of the arguments that are parameters

spynnaker.pyNN.models.defaults.**defaults**(*cls*)
Get the default parameters and state variables from the arguments to the __init__ method. This uses the decorators `default_parameters()` and `default_initial_values()` to determine the parameters and state variables respectively. If only one is specified, the other is assumed to be the remaining arguments. If neither are specified, it is assumed that all default arguments are parameters.

spynnaker.pyNN.models.defaults.**get_dict_from_init**(*init*, *skip=None*, *include=None*)

spynnaker.pyNN.models.pynn_population_common module

class spynnaker.pyNN.models.pynn_population_common.**PyNNPopulationCommon**(*spinnaker_control*, *size*, *label*, *constraints*, *model*, *structure*, *initial_values*, *additional_parameters=None*)
Bases: `object`

add_placement_constraint(*x*, *y*, *p=None*)
Add a placement constraint

Parameters

- **x** (`int`) – The x-coordinate of the placement constraint
- **y** (`int`) – The y-coordinate of the placement constraint
- **p** (`int`) – The processor ID of the placement constraint (optional)

all()
Iterator over cell IDs on all nodes.

can_record(*variable*)
Determine whether *variable* can be recorded from this population.

Note: This is supported by sPyNNaker8

conductance_based
True if the population uses conductance inputs

first_id

get(*parameter_names*, *gather=False*)
Get the values of a parameter for every local cell in the population.

Parameters **parameter_names** – Name of parameter. This is either a single string or a list of strings

Returns A single list of values (or possibly a single value) if paramter_names is a string, or a dict of these if parameter names is a list.

Return type str or list(str) or dict(str,str) or dict(str,list(str))

get_by_selector(*selector*, *parameter_names*)
Get the values of a parameter for the selected cell in the population.

Parameters

- **parameter_names** – Name of parameter. This is either a single string or a list of strings
- **selector** – a description of the subrange to accept. Or None for all. See: _selector_to_ids in SpiNNUtils.spinn_utilities.ranged.abstract_sized.py

Returns A single list of values (or possibly a single value) if paramter_names is a string or a dict of these if parameter names is a list.

Return type str or list(str) or dict(str,str) or dict(str,list(str))

get_spike_counts(*spikes*, *gather=True*)
Return the number of spikes for each neuron.

id_to_index(*id*)
Given the ID(s) of cell(s) in the Population, return its (their) index (order in the Population).

id_to_local_index(*cell_id*)
Given the ID(s) of cell(s) in the Population, return its (their) index (order in the Population), counting only cells on the local MPI node.

index_to_id(*index*)
Given the index (order in the Population) of cell(s) in the Population, return their ID(s)

inject(*current_source*)
Connect a current source to all cells in the Population.

label
The label of the population

last_id

local_size
The number of local cells

mark_no_changes()

positions
Return the position array for structured populations.

requires_mapping

set(*parameter*, *value=None*)
Set one or more parameters for every cell in the population.

param can be a dict, in which case value should not be supplied, or a string giving the parameter name, in which case value is the parameter value. value can be a numeric value, or list of such (e.g. for setting spike times):

```
p.set("tau_m", 20.0).
p.set({'tau_m':20, 'v_rest':-65})
```

Parameters

- **parameter** (*str or dict*) – the parameter to set
- **value** – the value of the parameter to set.

set_by_selector (*selector*, *parameter*, *value=None*)
Set one or more parameters for selected cell in the population.

param can be a dict, in which case value should not be supplied, or a string giving the parameter name, in which case value is the parameter value. value can be a numeric value, or list of such (e.g. for setting spike times):

```
p.set("tau_m", 20.0).
p.set({'tau_m':20, 'v_rest':-65})
```

Parameters

- **selector** – See RangedList.set_value_by_selector as this is just a pass through method
- **parameter** – the parameter to set
- **value** – the value of the parameter to set.

set_constraint (*constraint*)
Apply a constraint to a population that restricts the processor onto which its atoms will be placed.

set_mapping_constraint (*constraint_dict*)
Add a placement constraint - for backwards compatibility

Parameters constraint_dict (*dict (str, int)*) – A dictionary containing "x", "y" and optionally "p" as keys, and ints as values

set_max_atoms_per_core (*max_atoms_per_core*)
Supports the setting of this population's max atoms per core

Parameters max_atoms_per_core – the new value for the max atoms per core.

size
The number of neurons in the population

structure
Return the structure for the population.

spynnaker.pyNN.models.pynn_projection_common module

class spynnaker.pyNN.models.pynn_projection_common.**PyNNProjectionCommon**(*spinnaker_control, connector, synapse_dynamics_stdp, target, pre_synaptic_population, post_synaptic_population, rng, machine_time_step, user_max_delay, label, time_scale_factor*)

Bases: object

A container for all the connections of a given type (same synapse type and plasticity mechanisms) between two populations, together with methods to set parameters of those connections, including of plasticity mechanisms.

mark_no_changes()

requires_mapping

size(*gather=True*)

Return the total number of connections.

Parameters gather – If False, only get the number of connections locally. Which means nothing on SpiNNaker…

spynnaker.pyNN.models.recording_common module

class spynnaker.pyNN.models.recording_common.**RecordingCommon**(*population*)

Bases: object

Object to hold recording behaviour.

Parameters population – the population to record for

static pynn7_format(*data, ids, sampling_interval, data2=None*)

Module contents

spynnaker.pyNN.overridden_pacman_functions package

Submodules

spynnaker.pyNN.overridden_pacman_functions.graph_edge_filter module

class spynnaker.pyNN.overridden_pacman_functions.graph_edge_filter.**GraphEdgeFilter**

Bases: object

Removes graph edges that aren't required

spynnaker.pyNN.overridden_pacman_functions.graph_edge_weight_updater module

class spynnaker.pyNN.overridden_pacman_functions.graph_edge_weight_updater.**GraphEdgeWeightU**

Bases: object

Removes graph edges that aren't required

spynnaker.pyNN.overridden_pacman_functions.spynnaker_data_specification_writer module

class spynnaker.pyNN.overridden_pacman_functions.spynnaker_data_specification_writer.**Spynna**

Bases: spinn_front_end_common.interface.interface_functions.graph_data_specification_writer.GraphDataSpecificationWriter

Executes data specification generation for sPyNNaker

Module contents

spynnaker.pyNN.protocols package

Submodules

spynnaker.pyNN.protocols.munich_io_ethernet_protocol module

class spynnaker.pyNN.protocols.munich_io_ethernet_protocol.**MunichIoEthernetProtocol**

Bases: object

static disable_motor()

static disable_retina()

static enable_motor()

static enable_retina()

static laser_active_time(*active_time*)

static laser_frequency(*frequency*)

static laser_total_period(*total_period*)

static led_back_active_time(*active_time*)

static led_frequency(*frequency*)

static led_front_active_time(*active_time*)

static led_total_period(*total_period*)

static motor_0_leaky_velocity(*velocity*)

static motor_0_permanent_velocity(*velocity*)

static motor_1_leaky_velocity(*velocity*)

static motor_1_permanent_velocity(*velocity*)

static set_retina_transmission(*event_format*)

static speaker_active_time(*active_time*)

static speaker_frequency(*frequency*)

static speaker_total_period(*total_period*)

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol module

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**GET_RETINA_KEY_VALUE**(*payload*)

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**GET_RETINA_PAYLOAD_VALUE**(*payload*)

class spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**MunichIoSpiNNakerLinkProto**

Bases: object

Provides Multicast commands for the Munich SpiNNaker-Link protocol

Parameters

- **mode** – The mode of operation of the protocol
- **instance_key** – The optional instance key to use
- **uart_id** – The ID of the UART when needed

class MODES

Bases: enum.Enum

An enumeration.

BALL_BALANCER = 3

FREE = 5

MY_ORO_BOTICS = 4

PUSH_BOT = 1

RESET_TO_DEFAULT = 0

SPOMNIBOT = 2

add_payload_logic_to_current_output(*payload, time=None*)

add_payload_logic_to_current_output_key

bias_values(*bias_id, bias_value, time=None*)

bias_values_key

configure_master_key(*new_key, time=None*)

configure_master_key_key

disable_retina(*time=None*)

disable_retina_key

enable_disable_motor_key

generic_motor0_raw_output_leak_to_0(*pwm_signal, time=None*)

generic_motor0_raw_output_leak_to_0_key

generic_motor0_raw_output_permanent(*pwm_signal, time=None*)

generic_motor0_raw_output_permanent_key

generic_motor1_raw_output_leak_to_0 (*pwm_signal*, *time=None*)

generic_motor1_raw_output_leak_to_0_key

generic_motor1_raw_output_permanent (*pwm_signal*, *time=None*)

generic_motor1_raw_output_permanent_key

generic_motor_disable (*time=None*)

generic_motor_enable (*time=None*)

generic_motor_total_period (*time_in_ms*, *uart_id=0*, *time=None*)

generic_motor_total_period_key

instance_key
The key of this instance of the protocol

master_slave_key

master_slave_set_master_clock_active (*time=None*)

master_slave_set_master_clock_not_started (*time=None*)

master_slave_set_slave (*time=None*)

master_slave_use_internal_counter (*time=None*)

mode

poll_individual_sensor_continuously (*sensor_id*, *time_in_ms*, *time=None*)

poll_individual_sensor_continuously_key

poll_sensors_once (*sensor_id*, *time=None*)

poll_sensors_once_key

protocol_instance = 0

push_bot_laser_config_active_time (*active_time*, *time=None*)

push_bot_laser_config_active_time_key

push_bot_laser_config_total_period (*total_period*, *time=None*)

push_bot_laser_config_total_period_key

push_bot_laser_set_frequency (*frequency*, *time=None*)

push_bot_laser_set_frequency_key

push_bot_led_back_active_time (*active_time*, *time=None*)

push_bot_led_back_active_time_key

push_bot_led_front_active_time (*active_time*, *time=None*)

push_bot_led_front_active_time_key

push_bot_led_set_frequency (*frequency*, *time=None*)

push_bot_led_set_frequency_key

push_bot_led_total_period (*total_period*, *time=None*)

push_bot_led_total_period_key

push_bot_motor_0_leaking_towards_zero(*velocity*, *time=None*)

push_bot_motor_0_leaking_towards_zero_key

push_bot_motor_0_permanent(*velocity*, *time=None*)

push_bot_motor_0_permanent_key

push_bot_motor_1_leaking_towards_zero(*velocity*, *time=None*)

push_bot_motor_1_leaking_towards_zero_key

push_bot_motor_1_permanent(*velocity*, *time=None*)

push_bot_motor_1_permanent_key

push_bot_speaker_config_active_time(*active_time*, *time=None*)

push_bot_speaker_config_active_time_key

push_bot_speaker_config_total_period(*total_period*, *time=None*)

push_bot_speaker_config_total_period_key

push_bot_speaker_set_melody(*melody*, *time=None*)

push_bot_speaker_set_melody_key

push_bot_speaker_set_tone(*frequency*, *time=None*)

push_bot_speaker_set_tone_key

pwm_pin_output_timer_a_channel_0_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_a_channel_0_ratio_key

pwm_pin_output_timer_a_channel_1_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_a_channel_1_ratio_key

pwm_pin_output_timer_a_duration(*timer_period*, *time=None*)

pwm_pin_output_timer_a_duration_key

pwm_pin_output_timer_b_channel_0_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_b_channel_0_ratio_key()

pwm_pin_output_timer_b_channel_1_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_b_channel_1_ratio_key

pwm_pin_output_timer_b_duration(*timer_period*, *time=None*)

pwm_pin_output_timer_b_duration_key

pwm_pin_output_timer_c_channel_0_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_c_channel_0_ratio_key

pwm_pin_output_timer_c_channel_1_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_c_channel_1_ratio_key()

pwm_pin_output_timer_c_duration(*timer_period*, *time=None*)

pwm_pin_output_timer_c_duration_key

query_state_of_io_lines(*time=None*)

query_state_of_io_lines_key

remove_payload_logic_to_current_output(*payload*, *time=None*)

remove_payload_logic_to_current_output_key

reset_retina(*time=None*)

reset_retina_key

sensor_transmission_key(*sensor_id*)

static sent_mode_command()

True if the mode command has ever been requested by any instance

set_mode(*time=None*)

set_mode_key

set_output_pattern_for_payload(*payload*, *time=None*)

set_output_pattern_for_payload_key

set_payload_pins_to_high_impedance(*payload*, *time=None*)

set_payload_pins_to_high_impedance_key

set_retina_key(*new_key*, *time=None*)

set_retina_key_key

set_retina_transmission(*retina_key=<RetinaKey.NATIVE_128_X_128: 67108864>*, *retina_payload=None*, *time=None*)

Set the retina transmission key

Parameters

- **retina_key** – the new key for the retina
- **retina_payload** (`enum or None`) – the new payload for the set retina key command packet
- **time** – when to transmit this packet

Returns the command to send

Return type `spinn_front_end_common.utility_models.multi_cast_command.MultiCastCommand`

set_retina_transmission_key

turn_off_sensor_reporting(*sensor_id*, *time=None*)

turn_off_sensor_reporting_key

uart_id

class spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**RetinaKey**(*value*, *pixels*, *bits_per_coordinate*)

Bases: `enum.Enum`

An enumeration.

DOWNSAMPLE_16_X_16 = 268435456

DOWNSAMPLE_32_X_32 = 201326592

DOWNSAMPLE_64_X_64 = 134217728

FIXED_KEY = 0

NATIVE_128_X_128 = 67108864

bits_per_coordinate

n_neurons

pixels

class spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**RetinaPayload**(*value*, *n_payload_bytes*

Bases: enum.Enum

An enumeration.

ABSOLUTE_2_BYTE_TIMESTAMPS = 1073741824

ABSOLUTE_3_BYTE_TIMESTAMPS = 1610612736

ABSOLUTE_4_BYTE_TIMESTAMPS = 2147483648

DELTA_TIMESTAMPS = 536870912

EVENTS_IN_PAYLOAD = 0

NO_PAYLOAD = 0

n_payload_bytes

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**get_munich_d**(*key*)

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**get_munich_f**(*key*)

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**get_munich_i**(*key*)

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**get_push_bot_laser_led_speaker_**

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**get_push_bot_motor_i**(*key*)

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**get_retina_i**(*key*)

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**munich_key**(*I*, *F*, *D*)

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**munich_key_i**(*I*)

spynnaker.pyNN.protocols.munich_io_spinnaker_link_protocol.**munich_key_i_d**(*I*, *D*)

Module contents

class spynnaker.pyNN.protocols.**MunichIoEthernetProtocol**

Bases: object

static disable_motor()

static disable_retina()

static enable_motor()

static enable_retina()

static laser_active_time(*active_time*)

static laser_frequency(*frequency*)

static laser_total_period(*total_period*)

static led_back_active_time(*active_time*)

static led_frequency(*frequency*)

static led_front_active_time(*active_time*)

static led_total_period(*total_period*)

static motor_0_leaky_velocity(*velocity*)

static motor_0_permanent_velocity(*velocity*)

static motor_1_leaky_velocity(*velocity*)

static motor_1_permanent_velocity(*velocity*)

static set_retina_transmission(*event_format*)

static speaker_active_time(*active_time*)

static speaker_frequency(*frequency*)

static speaker_total_period(*total_period*)

class spynnaker.pyNN.protocols.**MunichIoSpiNNakerLinkProtocol**(*mode*, *instance_key=None*, *uart_id=0*)

Bases: `object`

Provides Multicast commands for the Munich SpiNNaker-Link protocol

Parameters

- **mode** – The mode of operation of the protocol
- **instance_key** – The optional instance key to use
- **uart_id** – The ID of the UART when needed

class MODES

Bases: `enum.Enum`

An enumeration.

BALL_BALANCER = 3

FREE = 5

MY_ORO_BOTICS = 4

PUSH_BOT = 1

RESET_TO_DEFAULT = 0

SPOMNIBOT = 2

add_payload_logic_to_current_output(*payload*, *time=None*)

add_payload_logic_to_current_output_key

bias_values(*bias_id*, *bias_value*, *time=None*)

bias_values_key

configure_master_key(*new_key*, *time=None*)

configure_master_key_key

disable_retina(*time=None*)

disable_retina_key

enable_disable_motor_key

generic_motor0_raw_output_leak_to_0 (*pwm_signal*, *time=None*)

generic_motor0_raw_output_leak_to_0_key

generic_motor0_raw_output_permanent (*pwm_signal*, *time=None*)

generic_motor0_raw_output_permanent_key

generic_motor1_raw_output_leak_to_0 (*pwm_signal*, *time=None*)

generic_motor1_raw_output_leak_to_0_key

generic_motor1_raw_output_permanent (*pwm_signal*, *time=None*)

generic_motor1_raw_output_permanent_key

generic_motor_disable (*time=None*)

generic_motor_enable (*time=None*)

generic_motor_total_period (*time_in_ms*, *uart_id=0*, *time=None*)

generic_motor_total_period_key

instance_key
The key of this instance of the protocol

master_slave_key

master_slave_set_master_clock_active (*time=None*)

master_slave_set_master_clock_not_started (*time=None*)

master_slave_set_slave (*time=None*)

master_slave_use_internal_counter (*time=None*)

mode

poll_individual_sensor_continuously (*sensor_id*, *time_in_ms*, *time=None*)

poll_individual_sensor_continuously_key

poll_sensors_once (*sensor_id*, *time=None*)

poll_sensors_once_key

protocol_instance = 0

push_bot_laser_config_active_time (*active_time*, *time=None*)

push_bot_laser_config_active_time_key

push_bot_laser_config_total_period (*total_period*, *time=None*)

push_bot_laser_config_total_period_key

push_bot_laser_set_frequency (*frequency*, *time=None*)

push_bot_laser_set_frequency_key

push_bot_led_back_active_time (*active_time*, *time=None*)

push_bot_led_back_active_time_key

push_bot_led_front_active_time (*active_time*, *time=None*)

push_bot_led_front_active_time_key

push_bot_led_set_frequency (*frequency*, *time=None*)

push_bot_led_set_frequency_key

push_bot_led_total_period (*total_period*, *time=None*)

push_bot_led_total_period_key

push_bot_motor_0_leaking_towards_zero (*velocity*, *time=None*)

push_bot_motor_0_leaking_towards_zero_key

push_bot_motor_0_permanent (*velocity*, *time=None*)

push_bot_motor_0_permanent_key

push_bot_motor_1_leaking_towards_zero (*velocity*, *time=None*)

push_bot_motor_1_leaking_towards_zero_key

push_bot_motor_1_permanent (*velocity*, *time=None*)

push_bot_motor_1_permanent_key

push_bot_speaker_config_active_time (*active_time*, *time=None*)

push_bot_speaker_config_active_time_key

push_bot_speaker_config_total_period (*total_period*, *time=None*)

push_bot_speaker_config_total_period_key

push_bot_speaker_set_melody (*melody*, *time=None*)

push_bot_speaker_set_melody_key

push_bot_speaker_set_tone (*frequency*, *time=None*)

push_bot_speaker_set_tone_key

pwm_pin_output_timer_a_channel_0_ratio (*timer_period*, *time=None*)

pwm_pin_output_timer_a_channel_0_ratio_key

pwm_pin_output_timer_a_channel_1_ratio (*timer_period*, *time=None*)

pwm_pin_output_timer_a_channel_1_ratio_key

pwm_pin_output_timer_a_duration (*timer_period*, *time=None*)

pwm_pin_output_timer_a_duration_key

pwm_pin_output_timer_b_channel_0_ratio (*timer_period*, *time=None*)

pwm_pin_output_timer_b_channel_0_ratio_key ()

pwm_pin_output_timer_b_channel_1_ratio (*timer_period*, *time=None*)

pwm_pin_output_timer_b_channel_1_ratio_key

pwm_pin_output_timer_b_duration (*timer_period*, *time=None*)

pwm_pin_output_timer_b_duration_key

pwm_pin_output_timer_c_channel_0_ratio (*timer_period*, *time=None*)

pwm_pin_output_timer_c_channel_0_ratio_key

pwm_pin_output_timer_c_channel_1_ratio (*timer_period*, *time=None*)

pwm_pin_output_timer_c_channel_1_ratio_key ()

pwm_pin_output_timer_c_duration (*timer_period*, *time=None*)

pwm_pin_output_timer_c_duration_key

query_state_of_io_lines (*time=None*)

query_state_of_io_lines_key

remove_payload_logic_to_current_output (*payload*, *time=None*)

remove_payload_logic_to_current_output_key

reset_retina (*time=None*)

reset_retina_key

sensor_transmission_key (*sensor_id*)

static sent_mode_command ()

True if the mode command has ever been requested by any instance

set_mode (*time=None*)

set_mode_key

set_output_pattern_for_payload (*payload*, *time=None*)

set_output_pattern_for_payload_key

set_payload_pins_to_high_impedance (*payload*, *time=None*)

set_payload_pins_to_high_impedance_key

set_retina_key (*new_key*, *time=None*)

set_retina_key_key

set_retina_transmission (*retina_key=<RetinaKey.NATIVE_128_X_128: 67108864>*, *retina_payload=None*, *time=None*)

Set the retina transmission key

Parameters

- **retina_key** – the new key for the retina
- **retina_payload** (`enum or None`) – the new payload for the set retina key command packet
- **time** – when to transmit this packet

Returns the command to send

Return type `spinn_front_end_common.utility_models.multi_cast_command.MultiCastCommand`

set_retina_transmission_key

turn_off_sensor_reporting (*sensor_id*, *time=None*)

turn_off_sensor_reporting_key

uart_id

class `spynnaker.pyNN.protocols.`**RetinaKey** (*value*, *pixels*, *bits_per_coordinate*)

Bases: `enum.Enum`

An enumeration.

DOWNSAMPLE_16_X_16 = 268435456

DOWNSAMPLE_32_X_32 = 201326592

DOWNSAMPLE_64_X_64 = 134217728

FIXED_KEY = 0

NATIVE_128_X_128 = 67108864

bits_per_coordinate

n_neurons

pixels

class spynnaker.pyNN.protocols.**RetinaPayload**(*value*, *n_payload_bytes*)
Bases: enum.Enum

An enumeration.

ABSOLUTE_2_BYTE_TIMESTAMPS = 1073741824

ABSOLUTE_3_BYTE_TIMESTAMPS = 1610612736

ABSOLUTE_4_BYTE_TIMESTAMPS = 2147483648

DELTA_TIMESTAMPS = 536870912

EVENTS_IN_PAYLOAD = 0

NO_PAYLOAD = 0

n_payload_bytes

spynnaker.pyNN.utilities package

Subpackages

spynnaker.pyNN.utilities.random_stats package

Submodules

spynnaker.pyNN.utilities.random_stats.abstract_random_stats module

class spynnaker.pyNN.utilities.random_stats.abstract_random_stats.**AbstractRandomStats**
Bases: object

Statistics about PyNN RandomDistribution objects

cdf(*dist*, *v*)
Return the cumulative distribution function value for the value v

high(*dist*)
Return the high cutoff value of the distribution, or None if the distribution is unbounded

low(*dist*)
Return the low cutoff value of the distribution, or None if the distribution is unbounded

mean(*dist*)
Return the mean of the distribution

ppf(*dist*, *p*)
Return the percent point function value for the probability p

std (*dist*)
Return the standard deviation of the distribution

var (*dist*)
Return the variance of the distribution

Module contents

class spynnaker.pyNN.utilities.random_stats.**AbstractRandomStats**
Bases: object

Statistics about PyNN RandomDistribution objects

cdf (*dist*, *v*)
Return the cumulative distribution function value for the value v

high (*dist*)
Return the high cutoff value of the distribution, or None if the distribution is unbounded

low (*dist*)
Return the low cutoff value of the distribution, or None if the distribution is unbounded

mean (*dist*)
Return the mean of the distribution

ppf (*dist*, *p*)
Return the percent point function value for the probability p

std (*dist*)
Return the standard deviation of the distribution

var (*dist*)
Return the variance of the distribution

spynnaker.pyNN.utilities.ranged package

Submodules

spynnaker.pyNN.utilities.ranged.spynnaker_ranged_dict module

class spynnaker.pyNN.utilities.ranged.spynnaker_ranged_dict.**SpynnakerRangeDictionary** (*size*, *de-faults=N*
Bases: spinn_utilities.ranged.range_dictionary.RangeDictionary

The Object is set up initially where every ID in the range will share the same value for each key. All keys must be of type str. The default Values can be anything including None.

Parameters

- **size** (*int*) – Fixed number of IDs / Length of lists
- **defaults** (*dict*) – Default dictionary where all keys must be str

list_factory (*size*, *value*, *key*)
Defines which class or subclass of RangedList to use

Main purpose is for subclasses to use a subclass or RangedList All parameters are pass through ones to the List constructor

Parameters

- **size** – Fixed length of the list
- **value** – value to given to all elements in the list
- **key** – The dict key this list covers.

Returns AbstractList in this case a RangedList

spynnaker.pyNN.utilities.ranged.spynnaker_ranged_list module

class spynnaker.pyNN.utilities.ranged.spynnaker_ranged_list.**SpynnakerRangedList**(*size=None, value=None, key=None, use_list_as_val*

Bases: spinn_utilities.ranged.ranged_list.RangedList

Parameters

- **size** – Fixed length of the list
- **value** – value to given to all elements in the list
- **key** – The dict key this list covers. This is used only for better Exception messages
- **use_list_as_value** – True if the value *is* a list

static as_list(*value, size, ids=None*)

Converts (if required) the value into a list of a given size. An exception is raised if value cannot be given size elements.

Note: This method can be extended to add other conversions to list in which case *is_list()* must also be extended.

Parameters value –

Returns value as a list

Raises Exception – if the number of values and the size do not match

static is_list(*value, size*)

Determines if the value should be treated as a list.

Note: This method can be extended to add other checks for list in which case *as_list()* must also be extended.

Module contents

class spynnaker.pyNN.utilities.ranged.**SpynnakerRangeDictionary**(*size, defaults=None*)

Bases: spinn_utilities.ranged.range_dictionary.RangeDictionary

The Object is set up initially where every ID in the range will share the same value for each key. All keys must be of type str. The default Values can be anything including None.

Parameters

- **size** (*int*) – Fixed number of IDs / Length of lists
- **defaults** (*dict*) – Default dictionary where all keys must be str

list_factory(*size*, *value*, *key*)

Defines which class or subclass of RangedList to use

Main purpose is for subclasses to use a subclass or RangedList All parameters are pass through ones to the List constructor

Parameters

- **size** – Fixed length of the list
- **value** – value to given to all elements in the list
- **key** – The dict key this list covers.

Returns AbstractList in this case a RangedList

class spynnaker.pyNN.utilities.ranged.**SpynnakerRangedList**(*size=None*, *value=None*, *key=None*, *use_list_as_value=False*)

Bases: spinn_utilities.ranged.ranged_list.RangedList

Parameters

- **size** – Fixed length of the list
- **value** – value to given to all elements in the list
- **key** – The dict key this list covers. This is used only for better Exception messages
- **use_list_as_value** – True if the value *is* a list

static as_list(*value*, *size*, *ids=None*)

Converts (if required) the value into a list of a given size. An exception is raised if value cannot be given size elements.

Note: This method can be extended to add other conversions to list in which case *is_list()* must also be extended.

Parameters **value** –

Returns value as a list

Raises **Exception** – if the number of values and the size do not match

static is_list(*value*, *size*)

Determines if the value should be treated as a list.

Note: This method can be extended to add other checks for list in which case *as_list()* must also be extended.

Submodules

spynnaker.pyNN.utilities.constants module

class spynnaker.pyNN.utilities.constants.**POPULATION_BASED_REGIONS**
Bases: `enum.Enum`

An enumeration.

CONNECTOR_BUILDER = 9

DIRECT_MATRIX = 10

NEURON_PARAMS = 1

POPULATION_TABLE = 3

PROFILING = 8

PROVENANCE_DATA = 7

RECORDING = 6

SYNAPSE_DYNAMICS = 5

SYNAPSE_PARAMS = 2

SYNAPTIC_MATRIX = 4

SYSTEM = 0

spynnaker.pyNN.utilities.extracted_data module

class spynnaker.pyNN.utilities.extracted_data.**ExtractedData**
Bases: `object`

Data holder for all synaptic data being extracted in parallel. @Chimp: play here to hearts content.

get (*projection*, *attribute*)
Allow getting data from a given projection and attribute

Parameters

- **projection** – the projection data was extracted from
- **attribute** – the attribute to retrieve

Returns the attribute data in a connection holder

set (*projection*, *attribute*, *data*)
Allow the addition of data from a projection and attribute.

Parameters

- **projection** – the projection data was extracted from
- **attribute** – the attribute to store
- **data** – attribute data in a connection holder

Return type None

spynnaker.pyNN.utilities.fake_HBP_Portal_machine_provider module

class spynnaker.pyNN.utilities.fake_HBP_Portal_machine_provider.**FakeHBPPortalMachineProvide**

Bases: object

create()

destroy()

get_machine_info()

wait_till_not_ready()

wait_until_ready()

spynnaker.pyNN.utilities.reports module

spynnaker.pyNN.utilities.running_stats module

class spynnaker.pyNN.utilities.running_stats.**RunningStats**

Bases: object

Keeps running statistics From: http://www.johndcook.com/blog/skewness_kurtosis/

add_item(*x*)

add_items(*mean*, *variance*, *n_items*)

mean

n_items

standard_deviation

variance

spynnaker.pyNN.utilities.spynnaker_connection_holder_generations module

class spynnaker.pyNN.utilities.spynnaker_connection_holder_generations.**SpYNNakerConnection**

Bases: object

Sets up connection holders for reports to use.

spynnaker.pyNN.utilities.spynnaker_failed_state module

class spynnaker.pyNN.utilities.spynnaker_failed_state.**SpynnakerFailedState**

Bases: *spynnaker.pyNN.spynnaker_simulator_interface.SpynnakerSimulatorInterface*, spinn_front_end_common.utilities.failed_state.FailedState, object

get_current_time()

get_distribution_to_stats()

get_pynn_NumpyRNG()

get_random_distribution()

has_reset_last

is_a_pynn_random(*thing*)

max_delay

min_delay

static reset(*annotations=None*)

set_number_of_neurons_per_core(*neuron_type*, *max_permitted*)

spynnaker.pyNN.utilities.spynnaker_neuron_network_specification_report module

class spynnaker.pyNN.utilities.spynnaker_neuron_network_specification_report.**SpYNNakerNeuro**

Bases: object

spynnaker.pyNN.utilities.spynnaker_synaptic_matrix_report module

class spynnaker.pyNN.utilities.spynnaker_synaptic_matrix_report.**SpYNNakerSynapticMatrixRepo**

Bases: object

Generate the synaptic matrix for reporting purposes

spynnaker.pyNN.utilities.utility_calls module

utility class containing simple helper methods

spynnaker.pyNN.utilities.utility_calls.**check_directory_exists_and_create_if_not**(*filename*)

Create a parent directory for a file if it doesn't exist

Parameters **filename** – The file whose parent directory is to be created

spynnaker.pyNN.utilities.utility_calls.**check_sampling_interval**(*sampling_interval*)

spynnaker.pyNN.utilities.utility_calls.**convert_param_to_numpy**(*param*, *no_atoms*)

Convert parameters into numpy arrays

Parameters

- **param** – the param to convert
- **no_atoms** – the number of atoms available for conversion of param

Return numpy.array the converted param in whatever format it was given

spynnaker.pyNN.utilities.utility_calls.**convert_to**(*value*, *data_type*)

Convert a value to a given data type

Parameters

- **value** – The value to convert
- **data_type** – The data type to convert to

Returns The converted data as a numpy data type

spynnaker.pyNN.utilities.utility_calls.**get_maximum_probable_value**(*dist*, *n_items*, *chance=0.01*)

Get the likely maximum value of a RandomDistribution given a number of draws `spynnaker.pyNN.utilities.utility_calls.`**`get_mean`**(*dist*)

Get the mean of a RandomDistribution

`spynnaker.pyNN.utilities.utility_calls.`**`get_minimum_probable_value`**(*dist*, *n_items*, *chance=0.01*)

Get the likely minimum value of a RandomDistribution given a number of draws

`spynnaker.pyNN.utilities.utility_calls.`**`get_n_bits`**(*n_values*)

Determine how many bits are required for the given number of values

`spynnaker.pyNN.utilities.utility_calls.`**`get_probability_within_range`**(*dist*, *lower*, *upper*)

Get the probability that a value will fall within the given range for a given RandomDistribution

`spynnaker.pyNN.utilities.utility_calls.`**`get_probable_maximum_selected`**(*n_total_trials*, *n_trials*, *selection_prob*, *chance=0.01*)

Get the likely maximum number of items that will be selected from a set of n_trials from a total set of n_total_trials with a probability of selection of selection_prob

`spynnaker.pyNN.utilities.utility_calls.`**`get_standard_deviation`**(*dist*)

Get the standard deviation of a RandomDistribution

`spynnaker.pyNN.utilities.utility_calls.`**`get_variance`**(*dist*)

Get the variance of a RandomDistribution

`spynnaker.pyNN.utilities.utility_calls.`**`high`**(*dist*)

Gets the high or max boundary value for this distribution

Could return None

`spynnaker.pyNN.utilities.utility_calls.`**`low`**(*dist*)

Gets the high or min boundary value for this distribution

Could return None

`spynnaker.pyNN.utilities.utility_calls.`**`read_in_data_from_file`**(*file_path*, *min_atom*, *max_atom*, *min_time*, *max_time*, *extra=False*)

Read in a file of data values where the values are in a format of: <time> <atom ID> <data value>

Parameters

- **`file_path`** – absolute path to a file containing the data
- **`min_atom`** – min neuron ID to which neurons to read in
- **`max_atom`** – max neuron ID to which neurons to read in
- **`min_time`** – min time slot to read neurons values of.
- **`max_time`** – max time slot to read neurons values of.

Returns a numpy array of (time stamp, atom ID, data value) `spynnaker.pyNN.utilities.utility_calls.`**`read_spikes_from_file`**(*file_path, min_atom=0, max_atom=inf, min_time=0, max_time=inf, split_value='\t'*)

Read spikes from a file formatted as: <time> <neuron ID>

Parameters

- **`file_path`** (`str`) – absolute path to a file containing spike values
- **`min_atom`** (`int`) – min neuron ID to which neurons to read in
- **`max_atom`** (`int`) – max neuron ID to which neurons to read in
- **`min_time`** (`int`) – min time slot to read neurons values of.
- **`max_time`** (`int`) – max time slot to read neurons values of.
- **`split_value`** (`str`) – the pattern to split by

Returns a numpy array with max_atom elements each of which is a list of spike times.

Return type numpy.array(int, int)

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

Update the seed to make it compatible with the rng algorithm

Module contents

1.1.1.2 Submodules

1.1.1.3 spynnaker.pyNN.abstract_spinnaker_common module

`class` `spynnaker.pyNN.abstract_spinnaker_common.`**`AbstractSpiNNakerCommon`**(*graph_label, database_socket_addresses, n_chips_required, timestep, max_delay, min_delay, hostname, user_extra_algorithm_xml_p user_extra_mapping_inputs= user_extra_algorithms_pre_r time_scale_factor=None, extra_post_run_algorithms=No extra_mapping_algorithms=No extra_load_algorithms=None, front_end_versions=None*)

Bases: `spinn_front_end_common.interface.abstract_spinnaker_base.AbstractSpinnakerBase`, *`spynnaker.pyNN.spynnaker_simulator_interface.SpynnakerSimulatorInterface`*

Main interface for neural code.

CONFIG_FILE_NAME = 'spynnaker.cfg'

add_application_vertex(*vertex_to_add*)

Parameters **vertex** – the vertex to add to the graph

Type ApplicationVertex

Return type None

Raises

- **ConfigurationException** – when both graphs contain vertices
- **PacmanConfigurationException** – If there is an attempt to add the same vertex more than once

add_population(*population*)
Called by each population to add itself to the list.

add_projection(*projection*)
Called by each projection to add itself to the list.

get_projections_data(*projection_to_attribute_map*)
Common data extractor for projection data. Allows fully exploitation of the ????

Parameters **projection_to_attribute_map** (*dict of projection with set of attributes*) – the projection to attributes mapping

Returns a extracted data object with get method for getting the data

Return type *spynnaker.pyNN.utilities.extracted_data.ExtractedData*

id_counter
Getter for id_counter, currently used by the populations.

Note: Maybe it could live in the pop class???

Returns

Return type int

max_delay
The maximum supported delay, in milliseconds.

min_delay
The minimum supported delay, in milliseconds.

static register_binary_search_path(*search_path*)
Register an additional binary search path for executables.

Parameters **search_path** – absolute search path for binaries

reset_number_of_neurons_per_core()

run(*run_time*)
Run the model created.

Parameters **run_time** – the time (in milliseconds) to run the simulation for

set_number_of_neurons_per_core(*neuron_type*, *max_permitted*)

stop(*turn_off_machine=None*, *clear_routing_tables=None*, *clear_tags=None*)

Parameters

- **turn_off_machine** (*bool*) – decides if the machine should be powered down after running the execution. Note that this powers down all boards connected to the BMP connections given to the transceiver
- **clear_routing_tables** (*bool*) – informs the tool chain if it should turn off the clearing of the routing tables
- **clear_tags** (*boolean*) – informs the tool chain if it should clear the tags off the machine at stop

Return type None

time_scale_factor
The multiplicative scaling from application time to real execution time.

Returns the time scale factor

1.1.1.4 spynnaker.pyNN.exceptions module

exception spynnaker.pyNN.exceptions.**DelayExtensionException**
Bases: spinn_front_end_common.utilities.exceptions.ConfigurationException

Raised when a delay extension vertex fails.

exception spynnaker.pyNN.exceptions.**FilterableException**
Bases: *spynnaker.pyNN.exceptions.SpynnakerException*

Raised when it is not possible to determine if an edge should be filtered.

exception spynnaker.pyNN.exceptions.**InvalidParameterType**
Bases: *spynnaker.pyNN.exceptions.SpynnakerException*

Raised when a parameter is not recognised.

exception spynnaker.pyNN.exceptions.**MemReadException**
Bases: *spynnaker.pyNN.exceptions.SpynnakerException*

Raised when the PyNN front end fails to read a certain memory region.

exception spynnaker.pyNN.exceptions.**SpynnakerException**
Bases: Exception

Superclass of all exceptions from the PyNN module.

exception spynnaker.pyNN.exceptions.**SynapseRowTooBigException**(*max_size*, *message*)
Bases: *spynnaker.pyNN.exceptions.SpynnakerException*

Raised when a synapse row is bigger than is allowed.PyNN

max_size
The maximum size allowed.

exception spynnaker.pyNN.exceptions.**SynapticBlockGenerationException**
Bases: spinn_front_end_common.utilities.exceptions.ConfigurationException

Raised when the synaptic manager fails to generate a synaptic block.

exception spynnaker.pyNN.exceptions.**SynapticBlockReadException**
Bases: spinn_front_end_common.utilities.exceptions.ConfigurationException

Raised when the synaptic manager fails to read a synaptic block or convert it into readable values.

exception spynnaker.pyNN.exceptions.**SynapticConfigurationException**
Bases: spinn_front_end_common.utilities.exceptions.ConfigurationException

Raised when the synaptic manager fails for some reason.

exception spynnaker.pyNN.exceptions.**SynapticMaxIncomingAtomsSupportException**
Bases: spinn_front_end_common.utilities.exceptions.ConfigurationException

Raised when a synaptic sublist exceeds the max atoms possible to be supported.

1.1.1.5 spynnaker.pyNN.spynnaker_external_device_plugin_manager module

class spynnaker.pyNN.spynnaker_external_device_plugin_manager.**SpynnakerExternalDevicePlugi**
Bases: object

User-level interface for the external device plugin manager.

static activate_live_output_for(*population*, *database_notify_host=None*, *database_notify_port_num=None*, *database_ack_port_num=None*, *board_address=None*, *port=None*, *host=None*, *tag=None*, *strip_sdp=True*, *use_prefix=False*, *key_prefix=None*, *prefix_type=None*, *message_type=<EIEIOType.KEY_32_BIT: 2>*, *right_shift=0*, *payload_as_time_stamps=True*, *notify=True*, *use_payload_prefix=True*, *payload_prefix=None*, *payload_right_shift=0*, *number_of_packets_sent_per_time_step=0*)

Output the spikes from a given population from SpiNNaker as they occur in the simulation.

Parameters

- **population** (*spynnaker.pyNN.models.pynn_population_common.PyNNPopulationCommon*) – The population to activate the live output for
- **database_notify_host** (*str*) – The hostname for the device which is listening to the database notification.
- **database_ack_port_num** (*int*) – The port number to which a external device will acknowledge that they have finished reading the database and are ready for it to start execution
- **database_notify_port_num** (*int*) – The port number to which a external device will receive the database is ready command
- **board_address** (*str*) – A fixed board address required for the tag, or None if any address is OK
- **key_prefix** (*int or None*) – the prefix to be applied to the key
- **prefix_type** – if the prefix type is 32 bit or 16 bit
- **message_type** – If the message is a EIEIO command message, or an EIEIO data message with 16 bit or 32 bit keys.
- **payload_as_time_stamps** –
- **right_shift** –
- **use_payload_prefix** –
- **notify** –
- **payload_prefix** –

- **payload_right_shift** –
- **number_of_packets_sent_per_time_step** –
- **port** (*int*) – The UDP port to which the live spikes will be sent. If not specified, the port will be taken from the "live_spike_port" parameter in the "Recording" section of the sPyNNaker configuration file.
- **host** (*str*) – The host name or IP address to which the live spikes will be sent. If not specified, the host will be taken from the "live_spike_host" parameter in the "Recording" section of the sPyNNaker configuration file.
- **tag** (*int*) – The IP tag to be used for the spikes. If not specified, one will be automatically assigned
- **strip_sdp** (*bool*) – Determines if the SDP headers will be stripped from the transmitted packet.
- **use_prefix** (*bool*) – Determines if the spike packet will contain a common prefix for the spikes

static activate_live_output_to(*population*, *device*)

Activate the output of spikes from a population to an external device. Note that all spikes will be sent to the device.

Parameters

- **population** (*spynnaker.pyNN.models.pynn_population_common.PyNNPopulationCommon*) – The pyNN population object from which spikes will be sent.
- **device** (*spynnaker.pyNN.models.pynn_population_common.PyNNPopulationCommon* or `pacman.model.graphs.application.ApplicationVertex`) – The pyNN population or external device to which the spikes will be sent.

static add_application_vertex(*vertex*)

static add_database_socket_address(*database_notify_host*, *database_notify_port_num*, *database_ack_port_num*)

static add_edge(*vertex*, *device_vertex*, *partition_id*)

Add an edge between two vertices (often a vertex and a external device) on a given partition.

Parameters

- **vertex** – the pre vertex to connect the edge from
- **device_vertex** – the post vertex to connect the edge to
- **partition_id** – the partition identifier for making nets

Return type None

static add_poisson_live_rate_control(*poisson_population*, *control_label_extension='_control'*, *receive_port=None*, *database_notify_host=None*, *database_notify_port_num=None*, *database_ack_port_num=None*, *notify=True*, *reserve_reverse_ip_tag=False*)

Add a live rate controller to a Poisson population.

Parameters

- **poisson_population** (*spynnaker.pyNN.models.pynn_population_common.PyNNPopulationCommon*) – The population to control
- **control_label_extension** (*str*) – An extension to add to the label of the Poisson source. Must match up with the equivalent in the SpynnakerPoissonControlConnection
- **receive_port** (*int*) – The port that the SpiNNaker board should listen on
- **database_notify_host** (*str*) – the hostname for the device which is listening to the database notification.
- **database_ack_port_num** (*int*) – the port number to which a external device will acknowledge that they have finished reading the database and are ready for it to start execution
- **database_notify_port_num** (*int*) – The port number to which a external device will receive the database is ready command
- **reserve_reverse_ip_tag** (*bool*) – True if a reverse ip tag is to be used, False if SDP is to be used (default)

static add_socket_address(*socket_address*)

Add a socket address to the list to be checked by the notification protocol.

Parameters **socket_address** – the socket address

Return type None:

static machine_time_step()

static time_scale_factor()

static update_live_packet_gather_tracker(*vertex_to_record_from, port, hostname, tag=None, board_address=None, strip_sdp=True, use_prefix=False, key_prefix=None, prefix_type=None, message_type=<EIEIOType.KEY_32_BIT: 2>, right_shift=0, payload_as_time_stamps=True, use_payload_prefix=True, payload_prefix=None, payload_right_shift=0, number_of_packets_sent_per_time_step=0, partition_id=None*)

Add an edge from a vertex to the live packet gatherer, builds as needed and has all the parameters for the creation of the live packet gatherer if needed.

1.1.1.6 spynnaker.pyNN.spynnaker_simulator_interface module

class spynnaker.pyNN.spynnaker_simulator_interface.**SpynnakerSimulatorInterface**

Bases: spinn_front_end_common.utilities.simulator_interface.SimulatorInterface

get_current_time()

get_distribution_to_stats()

get_pynn_NumpyRNG()

get_random_distribution()

has_reset_last

is_a_pynn_random (*thing*)

max_delay

min_delay

reset (*annotations=None*)

set_number_of_neurons_per_core (*neuron_type*, *max_permitted*)

1.1.1.7 Module contents

1.2 Submodules

1.3 spynnaker.gsyn_tools module

spynnaker.gsyn_tools.**check_gsyn** (*gsyn1*, *gsyn2*)

spynnaker.gsyn_tools.**check_path_gysn** (*path*, *n_neurons*, *runtime*, *gsyn*)

spynnaker.gsyn_tools.**check_sister_gysn** (*sister*, *n_neurons*, *runtime*, *gsyn*)

1.4 spynnaker.plot_utils module

spynnaker.plot_utils.**get_colour** ()

spynnaker.plot_utils.**grid** (*length*)

spynnaker.plot_utils.**heat_plot** (*data_sets*, *ylabel=None*, *title=None*)

spynnaker.plot_utils.**line_plot** (*data_sets*, *title=None*)

spynnaker.plot_utils.**plot_spikes** (*spikes*, *title='spikes'*)

Parameters **spikes** – Numpy array of spikes

1.5 spynnaker.spike_checker module

spynnaker.spike_checker.**synfire_multiple_lines_spike_checker** (*spikes*, *nNeurons*, *lines*, *wrap_around=True*)

Checks that there are the expected number of spike lines

Parameters

- **spikes** – The spikes
- **nNeurons** – Number of neurons
- **lines** – Expected number of lines
- **wrap_around** – If True the lines will wrap around when reaching the last neuron

spynnaker.spike_checker.**synfire_spike_checker** (*spikes*, *nNeurons*)

1.6 Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

A

C

D

E

H

K

L

M

N

Q

R

T

U

V

W