sPyNNaker Documentation

Release 6.0.0

Apr 12, 2021

Contents

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

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

Contents:

CHAPTER 1

spynnaker

1.1 spynnaker package

1.1.1 Subpackages

1.1.1.1 spynnaker.pyNN package

Subpackages

spynnaker.pyNN.connections package

Module contents

class spynnaker.pyNN.connections.**EthernetCommandConnection**(*translator*, *command_containers=None*, *local_host=None*, *local_port=19999*)

Bases: spinn_front_end_common.utilities.database.database_connection.DatabaseConnection

A connection that can send commands to a device at the start and end of a simulation

Parameters

- **translator** (AbstractEthernetTranslator) – A translator of multicast commands to device commands
- **command_containers** (*list* (*AbstractSendMeMulticastCommandsVertex*)) – A list of vertices that have commands to be sent at the start and end of simulation
- **local_host** (*str*) – The optional host to listen on for the start/resume message
- **local_port** (*int*) – The optional port to listen on for the stop/pause message

add_command_container(*command_container*)

Add a command container.

Parameters **command_container** (*AbstractSendMeMulticastCommandsVertex*) – A vertex that has commands to be sent at the start and end of simulation

class spynnaker.pyNN.connections.**EthernetControlConnection**(*translator*, *label*, *live_packet_gather_label*, *local_host=None*, *local_port=None*)

Bases: spinn_front_end_common.utilities.connections.live_event_connection.LiveEventConnection

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

Parameters

- **translator** (*AbstractEthernetTranslator*) – The translator of multicast to control commands
- **label** (*str*) – The label of the vertex to attach the translator to
- **live_packet_gather_label** (*str*) – The label of the LPG vertex that this control connection will listen to.
- **local_host** (*str*) – The optional host to listen on
- **local_port** (*int*) – The optional port to listen on

add_translator(*label*, *translator*)

Add another translator that routes via the LPG.

Parameters

- **label** (*str*) – The label of the vertex to attach the translator to
- **translator** (*AbstractEthernetTranslator*) – The translator of multicast to control commands

class spynnaker.pyNN.connections.**SpynnakerLiveSpikesConnection**(*receive_labels=None*, *send_labels=None*, *local_host=None*, *local_port=19999*, *live_packet_gather_label='LiveSpikeRec*

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(str)*) – Labels of population from which live spikes will be received.
- **send_labels** (*iterable(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

class spynnaker.pyNN.connections.**SpynnakerPoissonControlConnection**(*poisson_labels=None*, *local_host=None*, *local_port=19999*, *control_label_extension='_control'*)
Bases: spinn_front_end_common.utilities.connections.live_event_connection.LiveEventConnection

Parameters

- **poisson_labels** (*iterable(str)*) – Labels of Poisson populations to be controlled
- **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)
- **control_label_extension** (*str*) – The extra name added to the label of each Poisson source

add_init_callback(*label*, *init_callback*)
Add a callback to be called to initialise a vertex

Parameters

- **label** (*str*) – The label of the vertex to be notified about. Must be one of the vertices listed in the constructor
- **init_callback** (*callable(str, int, float, float) -> None*) – A function to be called to initialise the vertex. This should take as parameters the label of the vertex, the number of neurons in the population, the run time of the simulation in milliseconds, and the simulation timestep in milliseconds

add_pause_stop_callback(*label*, *pause_stop_callback*)
Add a callback for the pause and stop state of the simulation

Parameters

- **label** (*str*) – the label of the function to be sent
- **pause_stop_callback** (*callable(str, LiveEventConnection) -> None*) – A function to be called when the pause or stop message has been received. This function should take the label of the referenced vertex, and an instance of this class, which can be used to send events.

Return type None

add_poisson_label(*label*)

Parameters **label** (*str*) – The label of the Poisson source population.

add_receive_callback(*label*, *live_event_callback*, *translate_key=False*)
Add a callback for the reception of live events from a vertex

Parameters

- **label** (*str*) – The label of the vertex to be notified about. Must be one of the vertices listed in the constructor
- **live_event_callback** (*callable(str, int, list(int)) -> None*) – A function to be called when events are received. This should take as parameters the label of the vertex, the simulation timestep when the event occurred, and an array-like of atom IDs.
- **translate_key** (*bool*) – True if the key is to be converted to an atom ID, False if the key should stay a key

add_start_callback(*label*, *start_callback*)
Add a callback for the start of the simulation

Parameters

- **start_callback** (*callable(str, LiveEventConnection) -> None*) – A function to be called when the start message has been received. This function should take the label of the referenced vertex, and an instance of this class, which can be used to send events
- **label** (*str*) – the label of the function to be sent

add_start_resume_callback(*label*, *start_resume_callback*)
Add a callback for the start and resume state of the simulation

Parameters

- **label** (*str*) – the label of the function to be sent
- **start_resume_callback** (*callable(str, LiveEventConnection) -> None*) – A function to be called when the start or resume message has been received. This function should take the label of the referenced vertex, and an instance of this class, which can be used to send events.

Return type None

set_rate(*label*, *neuron_id*, *rate*)
Set the rate of a Poisson neuron within a Poisson source

Parameters

- **label** (*str*) – The label of the Population to set the rates of
- **neuron_id** (*int*) – The neuron ID to set the rate of
- **rate** (*float*) – The rate to set in Hz

set_rates (*label*, *neuron_id_rates*)
Set the rates of multiple Poisson neurons within a Poisson source

Parameters

- **label** (*str*) – The label of the Population to set the rates of
- **neuron_id_rates** (*list (tuple (int, float))*) – A list of tuples of (neuron ID, rate) to be set

spynnaker.pyNN.external_devices_models package

Subpackages

spynnaker.pyNN.external_devices_models.push_bot package

Subpackages

spynnaker.pyNN.external_devices_models.push_bot.control package

Module contents

class spynnaker.pyNN.external_devices_models.push_bot.control.**PushBotLifEthernet** (***kwargs*)
Bases: spynnaker.pyNN.external_devices_models.external_device_lif_control.ExternalDeviceLifControl

Leaky integrate and fire neuron with an exponentially decaying current input

Parameters

- **protocol** (MunichIoEthernetProtocol) – How to talk to the bot.
- **devices** (*iterable (AbstractMulticastControllableDevice)*) – The devices on the bot that we are interested in.
- **pushbot_ip_address** (*str*) – Where is the pushbot?
- **pushbot_port** (*int*) – (defaulted)
- **tau_m** (*float*) – LIF neuron parameter (defaulted)
- **cm** (*float*) – LIF neuron parameter (defaulted)
- **v_rest** (*float*) – LIF neuron parameter (defaulted)
- **v_reset** (*float*) – LIF neuron parameter (defaulted)
- **tau_syn_E** (*float*) – LIF neuron parameter (defaulted)
- **tau_syn_I** (*float*) – LIF neuron parameter (defaulted)
- **tau_refrac** (*float*) – LIF neuron parameter (defaulted)
- **i_offset** (*float*) – LIF neuron parameter (defaulted)
- **v** (*float*) – LIF neuron parameter (defaulted)
- **isyn_exc** (*float*) – LIF neuron parameter (defaulted)
- **isyn_inh** (*float*) – LIF neuron parameter (defaulted)

class spynnaker.pyNN.external_devices_models.push_bot.control.**PushBotLifSpinnakerLink**(***kwa*
Bases: spynnaker.pyNN.external_devices_models.external_device_lif_control.ExternalDeviceLifControl

Control module for a PushBot connected to a SpiNNaker Link

Parameters

- **protocol** (MunichIoSpiNNakerLinkProtocol) – How to talk to the bot.
- **devices** (*iterable*(AbstractMulticastControllableDevice)) – The devices on the bot that we are interested in.
- **tau_m** (*float*) – LIF neuron parameter (defaulted)
- **cm** (*float*) – LIF neuron parameter (defaulted)
- **v_rest** (*float*) – LIF neuron parameter (defaulted)
- **v_reset** (*float*) – LIF neuron parameter (defaulted)
- **tau_syn_E** (*float*) – LIF neuron parameter (defaulted)
- **tau_syn_I** (*float*) – LIF neuron parameter (defaulted)
- **tau_refrac** (*float*) – LIF neuron parameter (defaulted)
- **i_offset** (*float*) – LIF neuron parameter (defaulted)
- **v** (*float*) – LIF neuron parameter (defaulted)
- **isyn_exc** (*float*) – LIF neuron parameter (defaulted)
- **isyn_inh** (*float*) – LIF neuron parameter (defaulted)

spynnaker.pyNN.external_devices_models.push_bot.ethernet package

Module contents

class spynnaker.pyNN.external_devices_models.push_bot.ethernet.**PushBotEthernetDevice**(*protoco de- vice, uses_pa time_be*
Bases: spynnaker.pyNN.external_devices_models.abstract_multicast_controllable_device.AbstractMulticastControllableDevice

An arbitrary PushBot device

Parameters

- **protocol** (MunichIoEthernetProtocol) – The protocol instance to get commands from
- **device** (AbstractPushBotOutputDevice) – The Enum instance of the device to control
- **uses_payload** (*bool*) – 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

Return type *MunichIoEthernetProtocol*

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

Parameters **command_protocol** (`MunichIoSpiNNakerLinkProtocol`) – The protocol to use for this device

class `spynnaker.pyNN.external_devices_models.push_bot.ethernet.`**PushBotEthernetLaserDevice**(

Bases: `spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_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** (`PushBotLaser`) – The PushBotLaser value to control

- **protocol** (MunichIoEthernetProtocol) – 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.

Parameters **command_protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol to use for this 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)

class spynnaker.pyNN.external_devices_models.push_bot.ethernet.**PushBotEthernetLEDDevice** (*led, protocol, star… star… star… star… tim…*

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_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** (PushBotLED) – The PushBotLED parameter to control
- **protocol** (MunichIoEthernetProtocol) – 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.

Parameters **command_protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol to use for this 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)

class spynnaker.pyNN.external_devices_models.push_bot.ethernet.**PushBotEthernetMotorDevice**(*m*
p
t
c
t

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_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** (PushBotMotor) – a PushBotMotor value to indicate the motor to control
- **protocol** (MunichIoEthernetProtocol) – 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.

Parameters **command_protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol to use for this 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)

class spynnaker.pyNN.external_devices_models.push_bot.ethernet.**PushBotEthernetRetinaDevice**

Bases: spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_retina_device.AbstractPushBotRetinaDevice, spynnaker.pyNN.external_devices_models.abstract_ethernet_sensor.AbstractEthernetSensor

Parameters

- **protocol** (MunichIoEthernetProtocol) –
- **resolution** (PushBotRetinaResolution) –
- **pushbot_ip_address** –
- **pushbot_port** –
- **injector_port** –
- **local_host** –
- **local_port** –
- **retina_injector_label** –

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

Return type *SpynnakerLiveSpikesConnection*

Return type *PushBotRetinaConnection*

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

Return type str

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

Return type dict(str,Any)

get_n_neurons()

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

Return type int

get_translator()

Get a translator of multicast commands to Ethernet commands

Return type *AbstractEthernetTranslator*

class spynnaker.pyNN.external_devices_models.push_bot.ethernet.**PushBotEthernetSpeakerDevice**

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_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 Speaker of a PushBot

Parameters

- **speaker** (PushBotSpeaker) – The PushBotSpeaker value to control
- **protocol** (MunichIoEthernetProtocol) – The protocol instance to get commands from
- **start_active_time** – The "active time" to set at the start
- **start_total_period** – The "total period" to set at the start
- **start_frequency** – The "frequency" to set at the start
- **start_melody** – The "melody" 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.

Parameters **command_protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol to use for this 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)

class spynnaker.pyNN.external_devices_models.push_bot.ethernet.**PushBotRetinaConnection**(*retin push-bot_w res-o-lu-tion= <Ret naKe 6710 lo-cal_h lo-cal_p*

Bases: spynnaker.pyNN.connections.spynnaker_live_spikes_connection.SpynnakerLiveSpikesConnection

A connection that sends spikes from the PushBot retina to a spike injector in SpiNNaker. Note that this assumes a packet format of 16-bits per retina event.

Parameters

- **retina_injector_label** (*str*) –
- **pushbot_wifi_connection** (PushBotWIFIConnection) –
- **resolution** (PushBotRetinaResolution) –
- **local_host** (*str or None*) –
- **local_port** (*int or None*) –

class spynnaker.pyNN.external_devices_models.push_bot.ethernet.**PushBotTranslator**(*protocol, push-bot_wifi_conn*

Bases: spynnaker.pyNN.external_devices_models.abstract_ethernet_translator.AbstractEthernetTranslator

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

Parameters

- **protocol** (MunichIoEthernetProtocol) – The instance of the PushBot protocol to get keys from
- **pushbot_wifi_connection** (PushBotWIFIConnection) – 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** (*AbstractEIEIODataElement*) – A received multicast packet

Return type None

spynnaker.pyNN.external_devices_models.push_bot.ethernet.**get_pushbot_wifi_connection**(*remote_ re- mote_po*

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)

class spynnaker.pyNN.external_devices_models.push_bot.ethernet.**PushBotWIFIConnection**(*remote_ re- mote_po*

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 **SpinnmanIOException** – If there is an error setting up the communication channel

RECV_SIZE = 1024

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

get_receive_method()
Get the method that receives for this connection.

is_connected()
See 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, as a dotted string, e.g. *0.0.0.0*

Return type str

local_port
The local port to which the connection is bound.

Return type int

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 bytes

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, as a dotted string, or None if not connected remotely

Return type str or None

remote_port
The remote port to which the connection is connected, or None if not connected remotely

Return type int or None

send(*data*)
Send data down this connection

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

Raises SpinnmanIOException – If there is an error sending the data

spynnaker.pyNN.external_devices_models.push_bot.parameters package

Module contents

class spynnaker.pyNN.external_devices_models.push_bot.parameters.**PushBotLaser**
Bases: spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice

The properties of the laser device that may be set.

LASER_ACTIVE_TIME = 1
The active period for the laser (no larger than the total period)

LASER_FREQUENCY = 2
The frequency of the laser

LASER_TOTAL_PERIOD = 0
The total period for the laser

class spynnaker.pyNN.external_devices_models.push_bot.parameters.**PushBotLED**
Bases: spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice

The properties of the LED device that may be set.

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

The properties of the motor devices that may be set. The pushbot has two motors, 0 (left) and 1 (right).

MOTOR_0_LEAKY = 1
For motor 0, set a variable speed depending on time since event receive

MOTOR_0_PERMANENT = 0
For motor 0, set a particular speed

MOTOR_1_LEAKY = 3
For motor 1, set a variable speed depending on time since event receive

MOTOR_1_PERMANENT = 2
For motor 0, set a particular speed

class `spynnaker.pyNN.external_devices_models.push_bot.parameters.`**PushBotSpeaker**
Bases: `spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice`

The properties of the speaker device that may be set.

SPEAKER_ACTIVE_TIME = 1

SPEAKER_MELODY = 3

SPEAKER_TONE = 2

SPEAKER_TOTAL_PERIOD = 0

class `spynnaker.pyNN.external_devices_models.push_bot.parameters.`**PushBotRetinaResolution**
Bases: `enum.Enum`

Resolutions supported by the pushbot retina device

DOWNSAMPLE_16_X_16 = <RetinaKey.DOWNSAMPLE_16_X_16: 268435456>
Down sampled 64 (8×8) pixels to 1

DOWNSAMPLE_32_X_32 = <RetinaKey.DOWNSAMPLE_32_X_32: 201326592>
Down sampled 16 (4×4) pixels to 1

DOWNSAMPLE_64_X_64 = <RetinaKey.DOWNSAMPLE_64_X_64: 134217728>
Down sampled 4 (2×2) pixels to 1

NATIVE_128_X_128 = <RetinaKey.NATIVE_128_X_128: 67108864>
The native resolution

class `spynnaker.pyNN.external_devices_models.push_bot.parameters.`**PushBotRetinaViewer**(*resolutio port=0, dis- play_ma frame_ti de- cay_tim*

Bases: `threading.Thread`

A viewer for the pushbot's retina. This is a thread that can be launched in parallel with the control code.

Based on matplotlib

Parameters

- **resolution** (`PushBotRetinaResolution`) –
- **port** (*int*) –
- **display_max** (*float*) – Value of brightest pixel to show
- **frame_time_ms** (*int*) – How regularity to display frames (milliseconds)
- **decay_time_constant_ms** (*int*) – Time constant of pixel decay (milliseconds)

local_host

local_port

run()
How the viewer works when the thread is running.

spynnaker.pyNN.external_devices_models.push_bot.spinnaker_link package

Module contents

class spynnaker.pyNN.external_devices_models.push_bot.spinnaker_link.**PushBotSpiNNakerLinkLa**

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_laser_device.PushBotEthernetLaserDevice, pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex

The Laser of a PushBot

Parameters

- **laser** (PushBotLaser) – Which laser device to control
- **protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol instance to get commands from
- **spinnaker_link_id** (*int*) – The SpiNNakerLink that the PushBot is connected to
- **n_neurons** (*int*) – The number of neurons in the device
- **label** (*str*) – A label for the device
- **board_address** (*str or None*) – The IP address of the board that the device is connected to
- **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

default_parameters = {'board_address': None, 'label': None, 'n_neurons': 1, 'start_

class spynnaker.pyNN.external_devices_models.push_bot.spinnaker_link.**PushBotSpiNNakerLinkL**

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_led_device.PushBotEthernetLEDDevice, pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex

The LED of a PushBot

Parameters

- **led** (PushBotLED) – The LED device to control
- **protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol instance to get commands from
- **spinnaker_link_id** (*int*) – The SpiNNakerLink connected to
- **n_neurons** (*int*) – The number of neurons in the device
- **label** (*str*) – The label of the device
- **board_address** (*str or None*) – The IP address of the board that the device is connected to
- **start_active_time_front** (*int or None*) – The "active time" to set for the front LED at the start
- **start_active_time_back** (*int or None*) – The "active time" to set for the back LED at the start
- **start_total_period** (*int or None*) – The "total period" to set at the start
- **start_frequency** (*int or None*) – The "frequency" to set at the start

default_parameters = {'board_address': None, 'label': None, 'n_neurons': 1, 'start_

class spynnaker.pyNN.external_devices_models.push_bot.spinnaker_link.**PushBotSpiNNakerLinkMo**

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_motor_device.PushBotEthernetMotorDevice, pacman.

model.graphs.application.application_spinnaker_link_vertex.
ApplicationSpiNNakerLinkVertex

The motor of a PushBot

Parameters

- **motor** (PushBotMotor) – the motor to control
- **protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol used to control the device
- **spinnaker_link_id** (*int*) – The SpiNNakerLink connected to
- **n_neurons** (*int*) – The number of neurons in the device
- **label** (*str*) – The label of the device
- **board_address** (*str or None*) – The IP address of the board that the device is connected to

default_parameters = {'board_address': None, 'label': None, 'n_neurons': 1}

class spynnaker.pyNN.external_devices_models.push_bot.spinnaker_link.**PushBotSpiNNakerLinkRe**

Bases: spynnaker.pyNN.external_devices_models.push_bot.
abstract_push_bot_retina_device.AbstractPushBotRetinaDevice,
pacman.model.graphs.application.application_spinnaker_link_vertex.
ApplicationSpiNNakerLinkVertex

default_parameters = {'board_address': None, 'label': None}

routing_info(*routing_info*)

start_resume_commands

The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

class spynnaker.pyNN.external_devices_models.push_bot.spinnaker_link.**PushBotSpiNNakerLinkSp**

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.
push_bot_speaker_device.PushBotEthernetSpeakerDevice, pacman.
model.graphs.application.application_spinnaker_link_vertex.
ApplicationSpiNNakerLinkVertex

The speaker of a PushBot

Parameters

- **speaker** (PushBotSpeaker) – Which speaker device to control

- **protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol instance to get commands from
- **spinnaker_link_id** (*int*) – The SpiNNakerLink connected to
- **n_neurons** (*int*) – The number of neurons in the device
- **label** (*str*) – The label of the device
- **board_address** (*str or None*) – The IP address of the board that the device is connected to
- **start_active_time** – The "active time" to set at the start
- **start_total_period** – The "total period" to set at the start
- **start_frequency** – The "frequency" to set at the start
- **start_melody** – The "melody" to set at the start

default_parameters = {'board_address': None, 'label': None, 'n_neurons': 1, 'start_

Module contents

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

Bases: enum.Enum

Superclass of all output device descriptors

max_value

min_value

protocol_property

Return type property

send_type

Return type SendType

time_between_send

Return type int

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

An abstraction of a silicon retina attached to a SpiNNaker system.

Parameters

- **protocol** (MunichIoEthernetProtocol *or* MunichIoSpiNNakerLinkProtocol) –
- **resolution** (PushBotRetinaResolution) –

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.**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

Return type iterable(*AbstractMulticastControllableDevice*)

get_message_translator()
Get the translator of messages

Return type *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

Return type *SpynnakerLiveSpikesConnection*

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

Return type str

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

Return type dict(str,Any)

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

Return type int

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

Return type *AbstractEthernetTranslator*

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 (*AbstractEIEIODataElement*) – A received multicast packet

Return type None

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

Parameters

- **n_neurons** (*int*) – Number of neurons
- **fpga_link_id** (*int*) –
- **fpga_id** (*int*) –
- **board_address** (*str or None*) –
- **label** (*str or None*) –

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*, *n_steps_per_timestep*, *drop_late_spikes*, *splitter*)
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(AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type 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

Parameters

- **n_neurons** (*int*) – Number of neurons
- **spinnaker_link** (*int*) – The SpiNNaker link to which the cochlea is connected
- **label** (*str*) –
- **board_address** (*str*) –

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

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 partition 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.**MachineMunichMotorDevice**(*speed, sample_time, update_time, delay_time, delta_threshold, continue_if_not_different, label=None, constraints=None, app_vertex=None*)

Bases: pacman.model.graphs.machine.machine_vertex.MachineVertex, 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.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl, spinn_front_end_common.interface.provenance.provides_provenance_data_from_machine_impl.ProvidesProvenanceDataFromMachineImpl

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

Parameters

- **speed** (*int*) –
- **sample_time** (*int*) –
- **update_time** (*int*) –
- **delay_time** (*int*) –
- **delta_threshold** (*int*) –
- **continue_if_not_different** (*bool*) –
- **label** (*str*) –
- **constraints** –
- **app_vertex** –

INPUT_BUFFER_FULL_NAME = 'Times_the_input_buffer_lost_packets'
The name of the provenance item saying that packets were lost.

MOTOR_PARTITION_ID = 'MOTOR'

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_n_keys_for_partition(*_partition*)
Get the number of keys required by the given partition of edges.

Parameters **_partition** (*OutgoingEdgePartition*) – An partition that comes out of this vertex

Returns The number of keys required

Return type int

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)

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

Parameters **spec** (*DataSpecificationGenerator*) – The data specification to write to

resources_required
The resources required by the vertex

Return type ResourceContainer

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.abstract.abstract_one_app_one_machine_vertex.

AbstractOneAppOneMachineVertex, spinn_front_end_common.abstract_models.abstract_vertex_with_dependent_vertices.AbstractVertexWithEdgeToDependentVertices

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

Parameters

- **spinnaker_link_id** (*int*) – The SpiNNaker link to which the motor is connected
- **board_address** (*str or None*) –
- **speed** (*int*) –
- **sample_time** (*int*) –
- **update_time** (*int*) –
- **delay_time** (*int*) –
- **delta_threshold** (*int*) –
- **continue_if_not_different** (*bool*) –
- **label** (*str or None*) –

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)

edge_partition_identifiers_for_dependent_vertex(*vertex*)

Return the dependent edge identifiers for a particular dependent vertex.

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

Return type iterable(str)

class spynnaker.pyNN.external_devices_models.**MunichRetinaDevice**(*retina_key, spinnaker_link_id, position, label='MunichRetinaDevice', 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

An Omnibot silicon retina device.

Parameters

- **retina_key** (*int*) –
- **spinnaker_link_id** (*int*) – The SpiNNaker link to which the retina is connected

- **position** (*str*) – LEFT or RIGHT
- **label** (*str*) –
- **polarity** (*str*) – UP, DOWN or MERGED
- **board_address** (*str or None*) –

DOWN_POLARITY = 'DOWN'

LEFT_RETINA = 'LEFT'
Select the left retina

MERGED_POLARITY = 'MERGED'

RIGHT_RETINA = 'RIGHT'
Select the right retina

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 partition 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

Parameters **device** (*list (*AbstractMulticastControllableDevice*)*) –

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

Parameters **parameters** (*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** (*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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*float*) – The time to be advanced in one call to the update of this component

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

Return type list(int or float or list(int) or list(float) 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** (*list (list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

spynnaker.pyNN.extra_algorithms package

Subpackages

spynnaker.pyNN.extra_algorithms.splitter_components package

Module contents

class spynnaker.pyNN.extra_algorithms.splitter_components.**AbstractSpynnakerSplitterDelay**
Bases: object

Defines that a splitter is able to handle delays in some way.

Ideally the splitter and therefore the vertices it creates are able to handle some delay themselves and if more is needed have the ability to accept spikes from a DelayExtensionMachineVertex

MAX_SUPPORTED_DELAY_TICS = 16

accepts_edges_from_delay_vertex()
Confirms that the splitter's vertices can handle spikes coming from a `DelayExtensionMachineVertex`.

If this method returns False and the users ask for a delay larger than that allowed by *max_support_delay()*, an exception will be raised saying a different splitter is required.

Return type bool

max_support_delay()
returns the max amount of delay this post vertex can support.

Returns max delay supported in ticks

Return type int

class spynnaker.pyNN.extra_algorithms.splitter_components.**SplitterAbstractPopulationVertexS**
Bases: `pacman.model.partitioner_splitters.abstract_splitters.abstract_splitter_slice.AbstractSplitterSlice`, `spynnaker.pyNN.extra_algorithms.splitter_components.abstract_spynnaker_splitter_delay.AbstractSpynnakerSplitterDelay`

handles the splitting of the AbstractPopulationVertex via slice logic.

INVALID_POP_ERROR_MESSAGE = 'The vertex {} cannot be supported by the SplitterAbstract

SPLITTER_NAME = 'SplitterAbstractPopulationVertexSlice'

check_supported_constraints()

Raises PacmanInvalidParameterException – When partitioner constraints other than `MaxVertexAtomsConstraint` and `FixedVertexAtomsConstraint` are used.

constant_sdram(*vertex_slice*, *graph*)
returns the constant sdram used by the vertex slice.

Parameters

- **vertex_slice** (*Slice*) – the atoms to get constant sdram of
- **graph** (*ApplicationGraph*) – app graph

Return type ConstantSDRAM

cpu_cost(*vertex_slice*)
get cpu cost for a slice of atoms

Parameters vertex_slice (*Slice*) – slice of atoms

Return type CPUCyclesPerTickResourcer

create_machine_vertex(*vertex_slice*, *resources*, *label*, *remaining_constraints*)
creates a machine vertex

Parameters

- **vertex_slice** (*Slice*) – vertex slice
- **resources** (*ResourceTracker*) – resources
- **label** (*str*) – human readable label for machine vertex.
- **remaining_constraints** (*iterable(AbstractConstraint)*) – none partitioner constraints.

Returns machine vertex

Return type MachineVertex

dtcm_cost(*vertex_slice*)
get the dtcm cost for the slice of atoms

Parameters **vertex_slice** (*Slice*) – atom slice for dtcm calc.

Return type DTCMResource

get_in_coming_vertices(*edge*, *outgoing_edge_partition*, *src_machine_vertex*)
gets incoming vertices and their acceptable edge types

The input vertices are the ones that will serve as dest vertices for external edges. If more than one set of vertices match this description the splitter should use the ones used by the most general edge type/ down stream splitter.

Parameters

- **edge** (*ApplicationEdge*) – app edge
- **outgoing_edge_partition** (*OutgoingEdgePartition*) – outgoing edge partition
- **src_machine_vertex** (*MachineVertex*) – the src machine vertex

Returns dict of keys being machine vertices and values are a list of acceptable edge types.

Return type dict(MachineVertex,list(class))

get_out_going_vertices(*edge*, *outgoing_edge_partition*)
gets pre vertices and their acceptable edge types

The output vertices are the ones that will serve as source vertices for external edges. If more than one set of vertices match this description the splitter should use the ones used by the most general edge type/ down stream splitter.

Parameters

- **edge** (*ApplicationEdge*) – app edge
- **outgoing_edge_partition** (*OutgoingEdgePartition*) – outgoing edge partition

Returns dict of keys being machine vertices and values are a list of acceptable edge types.

Return type dict(MachineVertex,list(class))

get_resources_used_by_atoms(*vertex_slice*, *graph*)
gets the resources of a slice of atoms from a given app vertex.

Parameters

- **vertex_slice** (*Slice*) – the slice to find the resources of.
- **vertex_slice** – the slice
- **graph** (*MachineGraph*) – app graph

Returns Resource container.

Return type

Return type ResourceContainer

get_variable_sdram(*vertex_slice*)
returns the variable sdram from the recorder.

Parameters **vertex_slice** (*Slice*) – the atom slice for recording sdram

Returns the variable sdram used by the neuron recorder

Return type VariableSDRAM

set_governed_app_vertex(*app_vertex*)
Sets a app vertex to be governed by this splitter object. Once set it can't be reset

Parameters **app_vertex** (*ApplicationVertex*) – the app vertex to govern

Raises **PacmanConfigurationException** – if the app vertex has already been set.

class spynnaker.pyNN.extra_algorithms.splitter_components.**SplitterDelayVertexSlice**(*other_split*
Bases: pacman.model.partitioner_splitters.abstract_splitters.abstract_dependent_splitter.AbstractDependentSplitter

handles the splitting of the DelayExtensionVertex via slice logic.

splitter for delay extensions

Parameters **other_splitter** – the other splitter to split slices via.

DELAY_EXTENSION_SLICE_LABEL = 'DelayExtensionsMachineVertex for {} with slice {}'

DELAY_RECORDING_ERROR = 'The delay extensions does not record any variables. Therefore

ESTIMATED_CPU_CYCLES = 128

INVALID_POP_ERROR_MESSAGE = 'The vertex {} cannot be supported by the SplitterDelayVert

NEED_EXACT_ERROR_MESSAGE = 'DelayExtensionsSplitters need exact incoming slices. Please

SPLITTER_NAME = 'SplitterDelayVertexSlice'

WORDS_PER_ATOM = 27

check_supported_constraints()

Raises **PacmanInvalidParameterException** – When partitioner constraints other than MaxVertexAtomsConstraint and FixedVertexAtomsConstraint are used.

constant_sdram(*graph*)
returns the sdram used by the delay extension

Parameters **graph** (*ApplicationGraph*) – app graph

Return type ConstantSDRAM

cpu_cost(*vertex_slice*)
returns the cpu cost of the delay extension for a slice of atoms

Parameters **vertex_slice** (*Slice*) – slice of atoms

Return type CPUCyclesPerTickResource

create_machine_vertex(*vertex_slice*, *resource_tracker*, *label*, *remaining_constraints*, *graph*)
creates a delay extension machine vertex and adds to the tracker.

Parameters

- **vertex_slice** (*Slice*) – vertex slice
- **resource_tracker** (*ResourceTracker*) – resources
- **label** (*str*) – human readable label for machine vertex.
- **remaining_constraints** (*iterable(AbstractConstraint)*) – none partitioner constraints.
- **graph** (*ApplicationGraph*) – the app graph

Returns machine vertex

Return type *DelayExtensionMachineVertex*

create_machine_vertices (*resource_tracker*, *machine_graph*, *app_graph*)
method for specific splitter objects to use.

Parameters

- **resource_tracker** (*ResourceTracker*) – machine resources
- **machine_graph** (*MachineGraph*) – machine graph

Returns true if successful, false otherwise

Return type bool

dtcm_cost (*vertex_slice*)
returns the dtcm used by the delay extension slice.

Parameters **vertex_slice** (*Slice*) – vertex slice

Return type DTCMResource

get_in_coming_slices ()
A best effort prediction of the slices of the input vertices.

If this method is called after create_machine_vertices the splitter should return the actual slices of the input vertices. The second value returned is then always `True`

If this method is called before create_machine_vertices the splitter will have to make an estimate unless the actual slices it will use are already known. The second value returned is `True` if and only if the slices will not be changed.

The output vertices are the ones that will serve as source vertices for external edges. If more than one set of vertices match this description the splitter should use the ones used by the most general edge type/ down stream splitter.

Returns the slices incoming to this vertex, bool if estimate or exact

Return type tuple(list(Slice), bool)

get_in_coming_vertices (*edge*, *outgoing_edge_partition*, *src_machine_vertex*)
gets incoming vertices and their acceptable edge types

The input vertices are the ones that will serve as dest vertices for external edges. If more than one set of vertices match this description the splitter should use the ones used by the most general edge type/ down stream splitter.

Parameters

- **edge** (*ApplicationEdge*) – app edge
- **outgoing_edge_partition** (*OutgoingEdgePartition*) – outgoing edge partition
- **src_machine_vertex** (*MachineVertex*) – the src machine vertex

Returns dict of keys being machine vertices and values are a list of acceptable edge types.

Return type dict(MachineVertex,list(class))

get_out_going_slices ()
A best effort prediction of the slices of the output vertices.

If this method is called after create_machine_vertices the splitter should return the actual slices of the output vertices. The second value returned is then always `True`

If this method is called before create_machine_vertices the splitter will have to make an estimate unless the actual slices it will use are already known. The second value returned is `True` if and only if the slices will not be changed.

The output vertices are the ones that will serve as source vertices for external edges. If more than one set of vertices match this description the splitter should use the ones used by the most general edge type/down-stream splitter.

Returns list of Slices and bool of estimate or not

Return type tuple(list(Slice), bool)

get_out_going_vertices(*edge*, *outgoing_edge_partition*)
gets pre vertices and their acceptable edge types

The output vertices are the ones that will serve as source vertices for external edges. If more than one set of vertices match this description the splitter should use the ones used by the most general edge type/ down stream splitter.

Parameters

- **edge** (*ApplicationEdge*) – app edge
- **outgoing_edge_partition** (*OutgoingEdgePartition*) – outgoing edge partition

Returns dict of keys being machine vertices and values are a list of acceptable edge types.

Return type dict(MachineVertex,list(class))

get_resources_used_by_atoms(*vertex_slice*, *graph*)
ger res for a APV

Parameters

- **vertex_slice** – the slice
- **graph** – app graph

Return type ResourceContainer

machine_vertices_for_recording(*variable_to_record*)
Gets the machine vertices which are recording this variable.

Parameters **variable_to_record** (*str*) – the variable to get machine verts for.

Returns list of machine vertices

Return type iterable(MachineVertex)

reset_called()
reset the splitter to be as if it has not operated a splitting yet.

set_governed_app_vertex(*app_vertex*)
Sets a app vertex to be governed by this splitter object. Once set it can't be reset

Parameters **app_vertex** (*ApplicationVertex*) – the app vertex to govern

Raises **PacmanConfigurationException** – if the app vertex has already been set.

source_of_delay_vertex

class spynnaker.pyNN.extra_algorithms.splitter_components.**SpynnakerSplitterPartitioner**
Bases: pacman.operations.partition_algorithms.splitter_partitioner.SplitterPartitioner

a splitter partitioner that's bespoke for spynnaker vertices.

__call__(*app_graph*, *machine*, *plan_n_time_steps*, *pre_allocated_resources=None*)

Parameters

- **app_graph** (*ApplicationGraph*) – app graph
- **machine** (*Machine*) – machine
- **plan_n_time_steps** (*int*) – the number of time steps to run for
- **pre_allocated_resources** (*PreAllocatedResourceContainer or None*) – any pre-allocated res to account for before doing any splitting.

Return type tuple(MachineGraph, int)

Raises **PacmanPartitionException** – when it cant partition

create_machine_edge(*src_machine_vertex*, *dest_machine_vertex*, *common_edge_type*, *app_edge*, *machine_graph*, *app_outgoing_edge_partition*, *resource_tracker*)

Create the machine edge (if needed) and add it to the graph.

Some implementations of this method are able to detect that the requested edge is not actually needed so never create or add it.

Parameters

- **src_machine_vertex** (*MachineVertex*) – Src machine vertex of a edge
- **dest_machine_vertex** (*MachineVertex*) – Dest machine vertex of a edge
- **common_edge_type** (*MachineEdge*) – The edge type to build
- **app_edge** (*ApplicationEdge*) – The app edge this machine edge is to be associated with.
- **machine_graph** (*MachineGraph*) – Machine graph to add edge to.
- **app_outgoing_edge_partition** (*OutgoingEdgePartition*) – Partition
- **resource_tracker** (*ResourceTracker*) – The resource tracker.

class spynnaker.pyNN.extra_algorithms.splitter_components.**SpynnakerSplitterSelector**

Bases: spinn_front_end_common.interface.splitter_selectors.splitter_selector.SplitterSelector

splitter object selector that allocates splitters to app vertices that have not yet been given a splitter object. default for APV is the SplitterAbstractPopulationVertexSlice default for external device splitters are SplitterOneToOneLegacy default for the rest is the SpynnakerSplitterSliceLegacy.

Parameters **app_graph** (*ApplicationGraph*) – app graph

Raises **PacmanConfigurationException** – If a bad configuration is set

PROGRESS_BAR_NAME = 'Adding Splitter selectors where appropriate'

__call__(*app_graph*)

Add a splitter to every vertex that doesn't already have one.

Parameters **app_graph** (*ApplicationGraph*) – app graph

Raises **PacmanConfigurationException** – If a bad configuration is set

static abstract_pop_heuristic(*app_vertex*)

Assign the splitter for APV. Allows future overrides

Parameters **app_vertex** (*ApplicationGraph*) – app vertex

static external_fpga_link_heuristic(*app_vertex*)
Assign the splitter for FPGA link vertices. Allows future overrides

Parameters **app_vertex** (*ApplicationGraph*) – app vertex

static external_spinnaker_link_heuristic(*app_vertex*)
Assign the splitter for SpiNNaker link vertices. Allows future overrides

Parameters **app_vertex** (*ApplicationGraph*) – app vertex

static spike_source_array_heuristic(*app_vertex*)
Assign the splitter for SSA. Allows future overrides

Parameters **app_vertex** (*ApplicationGraph*) – app vertex

static spike_source_poisson_heuristic(*app_vertex*)
Assign the splitter for SSP. Allows future overrides

Parameters **app_vertex** (*ApplicationGraph*) – app vertex

class spynnaker.pyNN.extra_algorithms.splitter_components.**SpynnakerSplitterSliceLegacy**
Bases: pacman.model.partitioner_splitters.splitter_slice_legacy.SplitterSliceLegacy, spynnaker.pyNN.extra_algorithms.splitter_components.abstract_spynnaker_splitter_delay.AbstractSpynnakerSplitterDelay

Module contents

class spynnaker.pyNN.extra_algorithms.**AbstractMachineBitFieldRouterCompressor**
Bases: object

Algorithm that adds in regeneration of synaptic matrices to bitfield compression to spinn_front_end_common.interface.interface_functions.MachineBitFieldRouterCompressor

__call__(*routing_tables, transceiver, machine, app_id, provenance_file_path, machine_graph, placements, executable_finder, write_compressor_iobuf, produce_report, default_report_folder, target_length, routing_infos, time_to_try_for_each_iteration, use_timer_cut_off, machine_time_step, time_scale_factor, threshold_percentage, retry_count, executable_targets, read_expander_iobuf, compress_as_much_as_possible=False, provenance_data_objects=None*)
entrance for routing table compression with bit field

Parameters

- **routing_tables** (*MulticastRoutingTables*) – routing tables
- **transceiver** (*Transceiver*) – spinnman instance
- **machine** (*Machine*) – spinnMachine instance
- **app_id** (*int*) – app id of the application
- **provenance_file_path** (*str*) – file path for prov data
- **machine_graph** (*MachineGraph*) – machine graph
- **placements** (*Placements*) – placements on machine
- **executable_finder** (*ExecutableFinder*) – where are binaries are located
- **write_compressor_iobuf** (*bool*) – flag saying if read iobuf
- **produce_report** (*bool*) –

- **default_report_folder** (*str*) –
- **target_length** (*int*) –
- **routing_infos** (*RoutingInfo*) –
- **threshold_percentage** (*int*) – the percentage of bitfields to do on chip before its considered a success
- **retry_count** (*int or None*) – Number of times that the sorters should set of the compressions again. None for as much as needed
- **read_algorithm_iobuf** (*bool*) – flag saying if read iobuf
- **compress_as_much_as_possible** (*bool*) – flag asking if should compress as much as possible
- **read_expander_iobuf** (*bool*) – reads the synaptic expander iobuf.

Return type list(ProvenanceDataItem)

class spynnaker.pyNN.extra_algorithms.**DelaySupportAdder**

Bases: object

adds delay extension vertices into the APP graph as needed

Parameters

- **app_graph** (*ApplicationGraph*) – the app graph
- **machine_time_step** (*int*) – the machine time step
- **user_max_delay** (*int*) – the user defined max delay

Return type None

APP_DELAY_PROGRESS_BAR_TEXT = 'Adding delay extensions as required'

DELAYS_NOT_SUPPORTED_SPLITTER = 'The app vertex {} with splitter {} does not support d

END_USER_MAX_DELAY_DEFILING_ERROR_MESSAGE = 'The end user entered a max delay for which

INVALID_SPLITTER_FOR_DELAYS_ERROR_MSG = 'The app vertex {} with splitter {} does not s

NOT_SUPPORTED_DELAY_ERROR_MSG = 'The maximum delay {} for projection {} is not support

__call__(*app_graph*, *machine_time_step*, *user_max_delay*)

adds the delay extensions to the app graph, now that all the splitter objects have been set.

Parameters

- **app_graph** (*ApplicationGraph*) – the app graph
- **machine_time_step** (*int*) – the machine time step
- **user_max_delay** (*int*) – the user defined max delay

spynnaker.pyNN.extra_algorithms.**finish_connection_holders**(*application_graph*)

Finishes the connection holders after data has been generated within them, allowing any waiting callbacks to be called.

Parameters **application_graph** (*ApplicationGraph*) –

class spynnaker.pyNN.extra_algorithms.**GraphEdgeWeightUpdater**

Bases: object

Updates the weights of all edges.

__call__(*machine_graph*)

Parameters **machine_graph** (*MachineGraph*) – the machine_graph whose edges are to be updated

class spynnaker.pyNN.extra_algorithms.**OnChipBitFieldGenerator**
Bases: object

Executes bitfield and routing table entries for atom based routing.

__call__(*placements*, *app_graph*, *executable_finder*, *provenance_file_path*, *transceiver*, *write_bit_field_generator_iobuf*, *generating_bitfield_report*, *default_report_folder*, *machine_graph*, *routing_infos*, *generating_bit_field_summary_report*)
Loads and runs the bit field generator on chip.

Parameters

- **placements** (*Placements*) – placements
- **app_graph** (*ApplicationGraph*) – the app graph
- **executable_finder** (*ExecutableFinder*) – the executable finder
- **provenance_file_path** (*str*) – the path to where provenance data items are written
- **transceiver** (*Transceiver*) – the SpiNNMan instance
- **write_bit_field_generator_iobuf** (*bool*) – flag for report
- **generating_bitfield_report** (*bool*) – flag for report
- **default_report_folder** (*str*) – the file path for reports
- **machine_graph** (*MachineGraph*) – the machine graph
- **routing_infos** (*RoutingInfo*) – the key to edge map
- **generating_bit_field_summary_report** (*bool*) – whether to make summary report

class spynnaker.pyNN.extra_algorithms.**RedundantPacketCountReport**
Bases: object

__call__(*provenance_items*, *report_default_directory*)

Parameters

- **provenance_items** (*list* (*ProvenanceDataItem*)) –
- **report_default_directory** (*str*) –

class spynnaker.pyNN.extra_algorithms.**SpYNNakerConnectionHolderGenerator**
Bases: object

Sets up connection holders for reports to use.

__call__(*application_graph*)

Parameters **application_graph** (*ApplicationGraph*) – app graph

Returns the set of connection holders for after DSG generation

Return type dict(tuple(*ProjectionApplicationEdge*, *SynapseInformation*), *ConnectionHolder*)

class spynnaker.pyNN.extra_algorithms.**SpynnakerDataSpecificationWriter**
Bases: spinn_front_end_common.interface.interface_functions.graph_data_specification_writer.GraphDataSpecificationWriter

Executes data specification generation for sPyNNaker

__call__(*placements*, *hostname*, *report_default_directory*, *write_text_specs*, *machine*, *data_n_timesteps*)

Parameters

- **placements** (*Placements*) – placements of machine graph to cores
- **hostname** (*str*) – SpiNNaker machine name
- **report_default_directory** (*str*) – the location where reports are stored
- **write_text_specs** (*bool*) – True if the textual version of the specification is to be written
- **machine** (*Machine*) – the python representation of the SpiNNaker machine
- **data_n_timesteps** (*int*) – The number of timesteps for which data space will been reserved

Returns DSG targets (map of placement tuple and filename)

Return type tuple(DataSpecificationTargets, dict(tuple(int,int,int), int))

Raises **ConfigurationException** – If the DSG asks to use more SDRAM than is available.

class spynnaker.pyNN.extra_algorithms.**SpynnakerMachineBitFieldPairRouterCompressor**
Bases: spynnaker.pyNN.extra_algorithms.spynnaker_machine_bit_field_router_compressor.AbstractMachineBitFieldRouterCompressor

class spynnaker.pyNN.extra_algorithms.**SpynnakerMachineBitFieldUnorderedRouterCompressor**
Bases: spynnaker.pyNN.extra_algorithms.spynnaker_machine_bit_field_router_compressor.SpynnakerMachineBitFieldOrderedCoveringCompressor

DEPRACATED use SpynnakerMachineBitFieldOrderedCoveringCompressor

class spynnaker.pyNN.extra_algorithms.**SpYNNakerNeuronGraphNetworkSpecificationReport**
Bases: object

Produces a report describing the graph created from the neural populations and projections.

__call__(*report_folder*, *application_graph*)

Parameters

- **report_folder** (*str*) – the report folder to put figure into
- **application_graph** (*ApplicationGraph*) – the app graph

class spynnaker.pyNN.extra_algorithms.**SpYNNakerSynapticMatrixReport**
Bases: object

Generate the synaptic matrices for reporting purposes.

__call__(*report_folder*, *connection_holder*, *dsg_targets*)
Convert synaptic matrix for every application edge.

Parameters

- **report_folder** (*str*) – where to write the report
- **connection_holder** (*dict(tuple(ProjectionApplicationEdge, SynapseInformation), ConnectionHolder)*) – where the synaptic matrices are stored (possibly after retrieval from the machine)
- **dsg_targets** – used to check if connection holders are populated

`spynnaker.pyNN.extra_algorithms.`**`synapse_expander`**`(`*placements*, *transceiver*, *provenance_file_path*, *executable_finder*, *extract_iobuf*`)`

Run the synapse expander.

Note: Needs to be done after data has been loaded.

Parameters

- **`placements`** (*`Placements`*) – Where all vertices are on the machine.
- **`transceiver`** (*`Transceiver`*) – How to talk to the machine.
- **`provenance_file_path`** (*`str`*) – Where provenance data should be written.
- **`executable_finder`** (*`ExecutableFinder`*) – How to find the synapse expander binaries.
- **`extract_iobuf`** (*`bool`*) – flag for extracting iobuf

spynnaker.pyNN.model_binaries package

Module contents

This module contains no python code.

spynnaker.pyNN.models package

Subpackages

spynnaker.pyNN.models.abstract_models package

Module contents

`class` `spynnaker.pyNN.models.abstract_models.`**`AbstractAcceptsIncomingSynapses`**

Bases: `object`

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

Note: See *`verify_splitter()`*

`clear_connection_cache``()`

Clear the connection data stored in the vertex so far.

`get_connections_from_machine``(`*transceiver*, *placements*, *app_edge*, *synapse_info*`)`

Get the connections from the machine post-run.

Parameters

- **`transceiver`** (*`Transceiver`*) – How to read the connection data
- **`placements`** (*`Placements`*) – Where the connection data is on the machine

- **app_edge** (ProjectionApplicationEdge) – The edge for which the data is being read
- **synapse_info** (SynapseInformation) – The specific projection within the edge

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

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

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

verify_splitter (*splitter*)
Check that the splitter implements the API(s) expected by the SynapticMatrices

Any Vertex that implements this api should override ApplicationVertex.splitter method to also call this function

Parameters **splitter** (AbstractSpynnakerSplitterDelay) – the splitter

Raises **PacmanConfigurationException** – if the splitter is not an instance of Abstract-SpynnakerSplitterDelay

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

Indicates an application vertex class that can describe the units of some of its variables.

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

Parameters **variable** (*str*) – the variable to find units from

Returns the units as a string.

Return type str

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

Indicates that this object (an application vertex) has delay stages that are used to increase the space required for bitfields in *spynnaker.pyNN.utilities.bit_field_utilities.get_estimated_sdram_for_bit_field_region()*

n_delay_stages
The maximum number of delay stages required by any connection out of this delay extension vertex

Return type int

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

Indicates a class (a MachineVertex) that can describe the maximum rate that it sends spikes.

The *SynapticManager* assumes that all machine vertexes share the same synapse_information will have the same rates.

max_spikes_per_second ()
Get maximum expected number of spikes per second

Parameters **variable** (*str*) – the variable to find units from

Returns the units as a string.

Return type str

max_spikes_per_ts(*machine_time_step*)
Get maximum expected number of spikes per timestep

Parameters **machine_time_step** (*int*) – The timestime used in ms

Return type int

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

Indicates that this application vertex 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 or without `_init`
- **selector** (*None or slice or int or list(bool) or list(int)*) – a description of the subrange to accept, or None for all. See: `selector_to_ids()`

Returns A list or an Object which act like a list

Return type iterable

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

Parameters **selector** (*None or slice or int or list(bool) or list(int)*) – a description of the subrange to accept, or `None` for all. See: `selector_to_ids()`

Return type dict(str,Any)

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

Return type dict(str,Any)

initialize(*variable*, *value*, *selector=None*)
Set the initial value of one of the state variables of the neurons in this population.

Parameters

- **variable** (*str*) – The name of the variable to set
- **value** (*float or int or Any*) – The value of the variable to set

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

Return type iterable(str)

class spynnaker.pyNN.models.abstract_models.**AbstractPopulationSettable**

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

Indicates that some properties of this application vertex 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** (*None or slice or int or list(bool) or list(int)*) – See get_value_by_selector() as this is just a pass through method
- **key** (*str*) – the name of the parameter to change

Return type list(float or int)

n_atoms

" See n_atoms()

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

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

Parameters

- **selector** (*None or slice or int or list(bool) or list(int)*) – See RangedList.set_value_by_selector as this is just a pass through method
- **key** (*str*) – the name of the parameter to change
- **value** (*float or int or list(float) or list(int)*) – 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** (*Transceiver*) – the SpinnMan interface
- **placement** (*Placement*) – the placement of a vertex
- **vertex_slice** (*Slice*) – the slice of atoms for this vertex

Return type None

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

Parameters **key** (*str*) – the name of the property

Return type Any or float or int or list(float) or list(int)

set_value(*key*, *value*)

Set a property

Parameters

- **key** (*str*) – the name of the parameter to change
- **value** (*Any or float or int or list(float) or list(int)*) – the new value of the parameter to assign

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

Indicates a class (a MachineVertex) that has may need to run the SYNAPSE_EXPANDER aplx

Cores that do not use the synapse_manager should not implement this API even though their app vertex may hold a synapse_manager.

Note: This is *not* implemented by the *DelayExtensionMachineVertex*, which needs a different expander aplx

gen_on_machine()
True if the synapses of a the slice of this vertex should be generated on the machine.

Note: The typical implementation for this method will be to ask the app_vertex's synapse_manager

Return type bool

read_generated_connection_holders(*transceiver*, *placement*)
Fill in the connection holders

Note: The typical implementation for this method will be to ask the app_vertex's synapse_manager

Parameters

- **transceiver** (*Transceiver*) – How the data is to be read
- **placement** (*Placement*) – Where the data is on the machine

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

An object whose weight can be updated.

update_weight()
Update the weight.

spynnaker.pyNN.models.common package

Submodules

spynnaker.pyNN.models.common.recording_utils module

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

Parameters

- **buffer_max** (*int*) –
- **space_needed** (*int*) –
- **enable_buffered_recording** (*bool*) –

Return type int

spynnaker.pyNN.models.common.recording_utils.**get_data**(*transceiver*, *placement*, *region*, *region_size*)

Get the recorded data from a region.

Parameters

- **transceiver** (*Transceiver*) –
- **placement** (*Placement*) –
- **region** (*int*) –
- **region_size** (*int*) –

Return type tuple(bytearray, int)

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.

Parameters

- **n_machine_time_steps** (*int*) –
- **bytes_per_timestep** (*int*) –

Return type int

spynnaker.pyNN.models.common.recording_utils.**make_missing_string**(*missing*)

Parameters **missing** (*iterable(Placement)*) –

Return type str

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

Parameters

- **buffer_max** (*int*) –
- **space_needed** (*int*) –
- **enable_buffered_recording** (*bool*) –

Return type bool

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

Extracts numpy based data from a file

Parameters

- **no_loads** (*int*) – the number of numpy elements in the file
- **cache_file** (*FileIO*) – the file to extract from

Returns The extracted data

Return type ndarray

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*)

Clear the recorded data from the object

Parameters

- **variable** (*str*) – PyNN name of the variable
- **buffer_manager** (*BufferManager*) – the buffer manager object
- **placements** (*Placements*) – the placements object

Return type None

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

Get the recorded data

Parameters

- **variable** (*str*) – PyNN name of the variable
- **n_machine_time_steps** (*int*) –
- **placements** (*Placements*) –
- **buffer_manager** (*BufferManager*) –
- **machine_time_step** (*int*) – microseconds

Returns (data, recording_indices, sampling_interval)

Return type tuple(ndarray,list(int),float)

get_expected_n_rows (*n_machine_time_steps*, *sampling_rate*, *vertex*, *variable*)

Returns the number of expected rows for a given runtime

Parameters

- **n_machine_time_steps** (*int*) – map of vertex to steps.
- **sampling_rate** (*int*) – the sampling rate for this vertex
- **vertex** (*MachineVertex*) – the machine vertex
- **variable** (*str*) – the variable being recorded

Returns int the number of rows expected.

get_neuron_sampling_interval (*variable*)

Returns the current sampling interval for this variable

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

Returns Sampling interval in microseconds

Return type float

get_recordable_variables()
Returns a list of the PyNN names of variables this model is expected to collect

Return type list(str)

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

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

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

Parameters

- **variable** (*str*) – PyNN name of the variable
- **new_state** (*bool*) –
- **sampling_interval** (*int or None*) –
- **indexes** (*list or None*) – Which indices are to be recorded (or None for all)

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

Indicates that spikes can be recorded from this object.

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

Parameters

- **buffer_manager** (*BufferManager*) – the buffer manager object
- **placements** (*Placements*) – the placements object

Return type None

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

Parameters

- **placements** (*Placements*) – the placements object
- **buffer_manager** (*BufferManager*) – the buffer manager object
- **machine_time_step** (*int*) – the time step of the simulation, in microseconds

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

Return type ndarray(tuple(int,int))

get_spikes_sampling_interval()
Return the current sampling interval for spikes

Returns Sampling interval in microseconds

Return type float

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** (*int or None*) – The interval at which spikes are recorded. Must be a whole multiple of the timestep. None will be taken as the timestep.
- **indexes** (*list(int) or None*) – 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()

Return type int

get_n_cpu_cycles(*n_neurons*)

Return type int

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

Get the recorded spikes from the object

Parameters

- **label** (*str*) –
- **buffer_manager** (*BufferManager*) – the buffer manager object
- **region** (*int*) –
- **placements** (*Placements*) – the placements object
- **application_vertex** (*ApplicationVertex*) –
- **machine_time_step** (*int*) – the time step of the simulation, in microseconds
- **base_key_function** (*callable(MachineVertex,int)*) –

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

Return type ndarray(tuple(int,int))

record

Return type bool

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

Parameters

- **new_state** – bool

- **sampling_interval** – not supported functionality

class spynnaker.pyNN.models.common.**NeuronRecorder**(*allowed_variables*, *data_types*, *bitfield_variables*, *n_neurons*, *per_timestep_variables*, *per_timestep_datatypes*)

Bases: object

Parameters

- **allowed_variables** (*list* (*str*)) –
- **data_types** (*list* (*str*)) –
- **bitfield_variables** (*list* (*str*)) –
- **n_neurons** (*int*) –

PACKETS = 'packets-per-timestep'
packets-per-timestep

PACKETS_TYPE = 2
packets-per-timestep data type

SPIKES = 'spikes'
flag for spikes

check_indexes(*indexes*)

Parameters **indexes** (*list* (*int*)) –

static expected_rows_for_a_run_time(*n_machine_time_steps*, *sampling_rate*)
determines how many rows to see based off how long its ran for

Parameters

- **n_machine_time_steps** (*int*) – map of vertex to time steps
- **sampling_rate** (*float*) – the sampling rate for a given variable

Returns how many rows there should be.

Return type int

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

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

Parameters

- **variable** (*str*) – The PyNN variable name to get buffered sdram of
- **vertex_slice** (*Slice*) –
- **n_machine_time_steps** (*int*) – how many machine time steps to run for

Returns data size

Return type int

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

Parameters

- **variable** (*str*) – PyNN variable name
- **vertex_slice** (*Slice*) –

Returns

Return type int

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** (*str*) – PyNN variable name
- **vertex_slice** (*Slice*) –

Returns

Return type int

get_dtcm_usage_in_bytes(*vertex_slice*)

Parameters **vertex_slice** (*Slice*) –

Return type int

get_exact_static_sdram_usage(*vertex_slice*)
gets the exact sdram needed by the dsg region. :param ~pacman.model.graphs.common.Slice vertex_slice: :rtype: int

NOTE: does not take into account the struct that's being allocated by the c code

get_matrix_data(*label*, *buffer_manager*, *placements*, *application_vertex*, *variable*, *n_machine_time_steps*)
Read a data mapped to time and neuron IDs from the SpiNNaker machine and converts to required data types with scaling if needed.

Parameters

- **label** (*str*) – vertex label
- **buffer_manager** (*BufferManager*) – the manager for buffered data
- **placements** (*Placements*) – the placements object
- **application_vertex** (*ApplicationVertex*) –
- **variable** (*str*) – PyNN name for the variable (*V*, *gsy_inh*, etc.)
- **n_machine_time_steps** (*int*) –

Returns (data, recording_indices, sampling_interval)

Return type tuple(ndarray, list(int), float)

get_n_cpu_cycles(*n_neurons*)

Parameters **n_neurons** (*int*) –

Return type int

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

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

Returns Sampling interval in microseconds

Return type float

get_recordable_variables()

Return type iterable(str)

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** (*Slice*) –

Returns Highest possible overflow needed

Return type int

get_sdram_usage_in_bytes(*vertex_slice*)

Parameters **vertex_slice** (*Slice*) –

Return type int

get_spikes(*label*, *buffer_manager*, *placements*, *application_vertex*, *variable*, *machine_time_step*)

Read spikes mapped to time and neuron IDs from the SpiNNaker machine.

Parameters

- **label** (*str*) – vertex label
- **buffer_manager** (*BufferManager*) – the manager for buffered data
- **placements** (*Placements*) – the placements object
- **application_vertex** (*ApplicationVertex*) –
- **variable** (*str*) –
- **machine_time_step** (*int*) – microseconds

Returns

Return type ndarray(tuple(int,int))

get_static_sdram_usage(*vertex_slice*)

Parameters **vertex_slice** (*Slice*) –

Return type int

get_variable_sdram_usage(*vertex_slice*)

Parameters **vertex_slice** (*Slice*) –

Return type VariableSDRAM

is_recording(*variable*)

Parameters **variable** (*str*) –

Return type bool

recorded_ids_by_slice(*vertex_slice*)

Parameters **vertex_slice** (*Slice*) –

Return type list(int)

recorded_region_ids

Return type list(int)

recording_variables

Return type list(str)

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

Parameters

- **variable** (*str*) – PyNN variable name
- **new_state** (*bool*) –
- **sampling_interval** (*int*) –
- **indexes** (*iterable(int)*) –

write_neuron_recording_region (*spec*, *neuron_recording_region*, *vertex_slice*, *data_n_time_steps*)

recording data specification

Parameters

- **spec** (*DataSpecificationGenerator*) – dsg spec
- **neuron_recording_region** (*int*) – the recording region
- **vertex_slice** (*Slice*) – the vertex slice
- **data_n_time_steps** (*int*) – how many time steps to run this time

Return type None

class spynnaker.pyNN.models.common.**MultiSpikeRecorder**

Bases: object

get_dtcm_usage_in_bytes ()

Return type int

get_n_cpu_cycles (*n_neurons*)

Parameters n_neurons (*int*) –

Return type int

get_sdram_usage_in_bytes (*n_neurons*, *spikes_per_timestep*)

Return type AbstractSDRAM

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

Parameters

- **label** (*str*) –
- **buffer_manager** (*BufferManager*) – the buffer manager object
- **region** (*int*) –
- **placements** (*Placements*) –
- **application_vertex** (*ApplicationVertex*) –
- **machine_time_step** (*int*) – microseconds

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

Return type ndarray(tuple(int,int))

record

Return type bool

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

Parameters **key** (*str*) – the name of the property

Return type Any or float or int or list(float) or list(int)

set_value (*key*, *value*)

Set a property

Parameters

- **key** (*str*) – the name of the parameter to change
- **value** (*Any or float or int or list(float) or list(int)*) – the new value of the parameter to assign

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

Parameters

- **buffer_max** (*int*) –
- **space_needed** (*int*) –
- **enable_buffered_recording** (*bool*) –

Return type int

spynnaker.pyNN.models.common.**get_data** (*transceiver*, *placement*, *region*, *region_size*)

Get the recorded data from a region.

Parameters

- **transceiver** (*Transceiver*) –
- **placement** (*Placement*) –
- **region** (*int*) –
- **region_size** (*int*) –

Return type tuple(bytearray, int)

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

Parameters

- **buffer_max** (*int*) –
- **space_needed** (*int*) –
- **enable_buffered_recording** (*bool*) –

Return type bool

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.

Parameters

- **n_machine_time_steps** (*int*) –
- **bytes_per_timestep** (*int*) –

Return type int

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

Extracts numpy based data from a file

Parameters

- **no_loads** (*int*) – the number of numpy elements in the file
- **cache_file** (*FileIO*) – the file to extract from

Returns The extracted data

Return type ndarray

spynnaker.pyNN.models.neural_projections package

Subpackages

spynnaker.pyNN.models.neural_projections.connectors package

Module contents

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

Bases: object

Abstract class that all PyNN Connectors extend.

Parameters

- **safe** (*bool*) – if True, check that weights and delays have valid values. If False, this check is skipped. (NB: SpiNNaker always checks.)
- **callback** (*callable*) – Ignored
- **verbose** (*bool*) –
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or None to make one when needed

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

connect(*projection*)

Apply this connector to a projection.

> **Warning:** Do *not* call this! SpyNNaker does not work that way.

Parameters **projection** (Projection) –

Raises ***SpynnakerException*** – Always. Method not supported; profiled out.

could_connect (*_synapse_info*, *_pre_slice*, *_post_slice*)
Checks if a pre slice and a post slice could connect.

Typically used to determine if a Machine Edge should be created by checking that at least one of the indexes in the pre slice could over time connect to at least one of the indexes in the post slice.

> **Note:** This method should never return a false negative, but may return a false positives

Parameters

- **_pre_slice** (*Slice*) –
- **_post_slice** (*Slice*) –
- **_synapse_info** (SynapseInformation) –

Return type bool

create_synaptic_block (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)
Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum (*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

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

Parameters **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type float

get_n_connections_from_pre_vertex_maximum (*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum (*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_provenance_data (*synapse_info*)

Parameters **synapse_info** (SynapseInformation) –

Return type list(ProvenanceDataItem)

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

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

Parameters **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type float

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

Parameters weights (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type float

safe

Return type bool

set_projection_information(*machine_time_step*, *synapse_info*)

sets a connectors projection info :param int machine_time_step: machine time step :param SynapseInformation synapse_info: the synapse info

set_space(*space*)

Set the space object (allowed after instantiation).

Parameters space (*Space*) –

space

The space object (may be updated after instantiation).

Return type Space or None

synapse_info

The synapse_info object (may be updated after instantiation).

Return type synapse_info or None

use_direct_matrix(*synapse_info*)

Parameters synapse_info (SynapseInformation) –

Return type bool

verbose

Return type bool

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

Parameters

- **safe** (*bool*) –
- **callback** (*callable*) – Ignored
- **verbose** (*bool*) –

gen_connector_id

The ID of the connection generator on the machine.

Return type int

gen_connector_params(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Get the parameters of the on machine generation.

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(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

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –

Return type ndarray(uint32)

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

Parameters delays (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type int

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

Parameters delays (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type int

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

Parameters weights (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type int

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

Parameters weights (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type int

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

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –

Return type ndarray(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

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type bool

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

Bases: object

Connector that generates on machine and supports using PopulationViews

N_VIEWS_PARAMS = 4

get_view_lo_hi(*indexes*)

Get the low and high index values of the PopulationView

Parameters indexes (*list(int)*) – the indexes array of a PopulationView

Returns The low and high index values of the PopulationView

Return type uint, uint

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

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine, spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_ AbstractConnectorSupportsViewsOnMachine

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.
- **safe** (*bool*) – If True, check that weights and delays have valid values. If False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

allow_self_connections

Return type bool

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id

The ID of the connection generator on the machine.

Return type int

gen_connector_params(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Get the parameters of the on machine generation.

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes

The size of the connector parameters in bytes.

Return type int

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

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** (*ndarray(2, uint8)*) – An explicit boolean matrix that specifies the connections between the pre- and post-populations (see PyNN documentation). Must be 2D in practice.
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

get_delay_maximum(*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)

Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (*SynapseInformation*) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (*SynapseInformation*) –

Return type int

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

Parameters **synapse_info** (*SynapseInformation*) –

Return type float

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.

Note: If you get TypeError in Python 3 see: https://github.com/INCF/csa/issues/10

Parameters

- **cset** (*csa.connset.CSet*) – A description of the connection set between populations
- **safe** (*bool*) – If True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

  Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

Raises **ImportError** – if the *csa* library isn't present; it's tricky to install in some environments so we don't force it to be present unless you want to actually use this class.

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

get_delay_maximum(*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)

Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (`SynapseInformation`) –

Return type int

get_weight_maximum(*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (`SynapseInformation`) –

Return type float

show_connection_set(*n_pre_neurons*, *n_post_neurons*)

Parameters

- **n_pre_neurons** (*int*) –
- **n_post_neurons** (*int*) –

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** (*str*) – 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** (*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.
- **safe** (*bool*) – if `True`, check that weights and delays have valid values. If `False`, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **n_connections** (*int or None*) – The number of efferent synaptic connections per neuron.
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or `None` to make one when needed.
- **callback** (*callable*) –

allow_self_connections

Return type bool

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

d_expression

The distance expression.

Return type str

get_delay_maximum(*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)

Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_weight_maximum(*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (SynapseInformation) –

Return type float

set_projection_information(*machine_time_step*, *synapse_info*)

sets a connectors projection info :param int machine_time_step: machine time step :param SynapseInformation synapse_info: the synapse info

class spynnaker.pyNN.models.neural_projections.connectors.**FixedNumberPostConnector**(*n*, *allow_self_c*, *safe=True*, *verbose=False*, *with_replac*, *rng=None*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine, spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_o AbstractConnectorSupportsViewsOnMachine

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.
- **safe** (*bool*) – Whether to check that weights and delays have valid values; if `False`, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **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.
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or `None` to make one when needed.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

allow_self_connections

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id

The ID of the connection generator on the machine.

Return type int

gen_connector_params(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Get the parameters of the on machine generation.

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes

The size of the connector parameters in bytes.

Return type int

get_delay_maximum(*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

set_projection_information(*machine_time_step*, *synapse_info*)
sets a connectors projection info :param int machine_time_step: machine time step :param SynapseInformation synapse_info: the synapse info

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

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

`spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_o`
`AbstractConnectorSupportsViewsOnMachine`

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.
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If `False`, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **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.
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or None to make one when needed
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

allow_self_connections

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (`AbstractSynapseType`) –
- **synapse_info** (`SynapseInformation`) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id

The ID of the connection generator on the machine.

Return type int

gen_connector_params (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Get the parameters of the on machine generation.

Parameters

- **pre_slices** (*list (Slice)*) –
- **post_slices** (*list (Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes

The size of the connector parameters in bytes.

Return type int

get_delay_maximum (*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum (*synapse_info*)

Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum (*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list (int) or list (float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum (*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

set_projection_information(*machine_time_step*, *synapse_info*)
sets a connectors projection info :param int machine_time_step: machine time step :param SynapseInformation synapse_info: the synapse info

class spynnaker.pyNN.models.neural_projections.connectors.**FixedProbabilityConnector**(*p_connec al-low_self_ safe=Tru ver-bose=Fal rng=None call-back=Nor*

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine, spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_c AbstractConnectorSupportsViewsOnMachine

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

Parameters

- **p_connect** (*float*) – a value 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.
- **safe** (*bool*) – If True, check that weights and delays have valid values. If False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or None to make one when needed
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)
Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id
The ID of the connection generator on the machine.

Return type int

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

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum (*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum (*post_vertex_slice, synapse_info, 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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_weight_maximum(*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (SynapseInformation) –

Return type float

p_connect

class spynnaker.pyNN.models.neural_projections.connectors.**FromFileConnector**(*file, distributed=False, safe=True, callback=None, verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.from_list_connector.FromListConnector

Make connections according to a list read from a file.

Parameters

- **file** (*str or FileIO*) – Either an open file object or the filename of a file containing a list of connections, in the format required by *FromListConnector*. Column headers, if included in the file, must be specified using a list or tuple, e.g.:

```
# columns = ["i", "j", "weight", "delay", "U", "tau_rec"]
```

  Note that the header requires # at the beginning of the line.

- **distributed** (*bool*) – Basic pyNN says:

  if this is True, then each node will read connections from a file called filename.x, where x is the MPI rank. This speeds up loading connections for distributed simulations.

Note: Always leave this as `False` with sPyNNaker, which is not MPI-based.

- **safe** (*bool*) – Whether to check that weights and delays have valid values. If `False`, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

get_reader (*file*)

Get a file reader object using the PyNN methods.

Returns A pynn StandardTextFile or similar

Return type StandardTextFile

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

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

Make connections according to a list.

Parameters

- **conn_list** (*ndarray or list(tuple(int,int,..))*) – A numpy array or a list of tuples, one tuple for each connection. Each tuple should contain:

```
(pre_idx, post_idx, p1, p2, ..., pn)
```

  where `pre_idx` is the index (i.e. order in the Population, not the ID) of the presynaptic neuron, `post_idx` is the index of the postsynaptic neuron, and `p1`, `p2`, etc. are the synaptic parameters (e.g., weight, delay, plasticity parameters). All tuples/rows must have the same number of items.
- **safe** (*bool*) – if `True`, check that weights and delays have valid values. If `False`, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **column_names** (*None or tuple(str) or list(str)*) – the names of the parameters `p1`, `p2`, etc. If not provided, it is assumed the parameters are `weight, delay` (for backwards compatibility).
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

column_names

The names of the columns in the array after the first two. Of particular interest is whether `weight` and `delay` columns are present.

Return type list(str)

conn_list

The connection list.

Return type ndarray

could_connect (*_synapse_info*, *_pre_slice*, *_post_slice*)

Checks if a pre slice and a post slice could connect.

Typically used to determine if a Machine Edge should be created by checking that at least one of the indexes in the pre slice could over time connect to at least one of the indexes in the post slice.

Note: This method should never return a false negative, but may return a false positives

Parameters

- **_pre_slice** (*Slice*) –
- **_post_slice** (*Slice*) –
- **_synapse_info** (`SynapseInformation`) –

Return type bool

create_synaptic_block (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (`AbstractSynapseType`) –
- **synapse_info** (`SynapseInformation`) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

get_delay_maximum (*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (`SynapseInformation`) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

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

Parameters **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type float

get_extra_parameter_names()
Getter for the names of the extra parameters.

Return type list(str)

get_extra_parameters()
Getter for the extra parameters. Excludes `weight` and `delay` columns.

Returns The extra parameters

Return type ndarray

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

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_hi** (*int*) –
- **post_hi** (*int*) –

Return type int

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

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

Parameters **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type float

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

Parameters **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type float

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

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** (*str*) – 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; the indices will be given as variables i and j when the expression is evaluated.
- **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.
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or None to make one when needed.
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

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.

Return type bool

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)
Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

index_expression
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.

Return type str

class spynnaker.pyNN.models.neural_projections.connectors.**KernelConnector**(*shape_pre, shape_post, shape_kernel, weight_kernel=None, delay_kernel=None, shape_common=None, pre_sample_steps_in_po[...] pre_start_coords_in_pos[...] post_sample_steps_in_p[...] post_start_coords_in_pr[...] safe=True, space=None, verbose=False, callback=None*)

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** (*list(int) or tuple(int,int)*) – 2D shape of the pre population (rows/height, cols/width, usually the input image shape)
- **shape_post** (*list(int) or tuple(int,int)*) – 2D shape of the post population (rows/height, cols/width)
- **shape_kernel** (*list(int) or tuple(int,int)*) – 2D shape of the kernel (rows/height, cols/width)
- **weight_kernel** (*ndarray or NumpyRNG or int or float or list(int) or list(float) or None*) – (optional) 2D matrix of size shape_kernel describing the weights
- **delay_kernel** (*ndarray or NumpyRNG or int or float or list(int) or list(float) or None*) – (optional) 2D matrix of size shape_kernel describing the delays
- **shape_common** (*list(int) or tuple(int,int) or None*) – (optional) 2D shape of common coordinate system (for both pre and post, usually the input image sizes)
- **pre_sample_steps_in_post** (*None or list(int) or tuple(int,int)*) – (optional) Sampling steps/jumps for pre pop <=> (stepX, stepY)
- **pre_start_coords_in_post** (*None or list(int) or tuple(int,int)*) – (optional) Starting row/col for pre sampling <=> (offX, offY)
- **post_sample_steps_in_pre** (*None or list(int) or tuple(int,int)*) – (optional) Sampling steps/jumps for post pop <=> (stepX, stepY)
- **post_start_coords_in_pre** (*None or list(int) or tuple(int,int)*) – (optional) Starting row/col for post sampling <=> (offX, offY)
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If `False`, this check is skipped.
- **space** (*Space*) – Currently ignored; for future compatibility.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **callback** (*callable*) – (ignored)

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (`AbstractSynapseType`) –

- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id
The ID of the connection generator on the machine.

Return type int

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

Parameters

- **pre_slices** (*list* (*Slice*)) –
- **post_slices** (*list* (*Slice*)) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(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

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list* (*int*) *or list* (*float*)) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –

Return type ndarray(uint32)

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

Parameters **delays** (*ndarray or NumpyRNG or int or float or list* (*int*) *or list* (*float*)) –

Return type int

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

Parameters **delays** (*ndarray or NumpyRNG or int or float or list* (*int*) *or list* (*float*)) –

Return type int

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

Parameters **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type int

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

Parameters **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type int

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

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –

Return type ndarray(uint32)

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

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

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine, spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_ AbstractConnectorSupportsViewsOnMachine

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

- **n** (*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.
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or None to make one when needed.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

create_synaptic_block (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)
Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (Slice) –
- **post_vertex_slice** (Slice) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id
The ID of the connection generator on the machine.

Return type int

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

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (Slice) –
- **post_vertex_slice** (Slice) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum (*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum (*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_rng_next(*num_synapses*, *prob_connect*)

Get the required RNGs

Parameters

- **num_synapses** (*int*) – The number of synapses to make random numbers for in this call
- **prob_connect** (*list(float)*) – The probability of connection

Return type ndarray

get_weight_maximum(*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (SynapseInformation) –

Return type float

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

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine, spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_on_machine.AbstractConnectorSupportsViewsOnMachine

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

Parameters

- **safe** (*bool*) – If True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

could_connect (*_synapse_info*, *_pre_slice*, *_post_slice*)
Checks if a pre slice and a post slice could connect.

Typically used to determine if a Machine Edge should be created by checking that at least one of the indexes in the pre slice could over time connect to at least one of the indexes in the post slice.

Note: This method should never return a false negative, but may return a false positives

Parameters

- **_pre_slice** (*Slice*) –
- **_post_slice** (*Slice*) –
- **_synapse_info** (SynapseInformation) –

Return type bool

create_synaptic_block (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)
Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id
The ID of the connection generator on the machine.

Return type int

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

Parameters

- **pre_slices** (*list (Slice)*) –
- **post_slices** (*list (Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum (*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum (*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list (int) or list (float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum (*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

use_direct_matrix(*synapse_info*)

Parameters **synapse_info** (SynapseInformation) –

Return type bool

class spynnaker.pyNN.models.neural_projections.connectors.**SmallWorldConnector**(*degree, rewiring, allow_self_connections=True, n_connections=None, rng=None, safe=True, callback=None, verbose=False*)

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

A connector that uses connection statistics based on the Small World network connectivity model.

Note: This is typically used from a population to itself.

Parameters

- **degree** (*float*) – the region length where nodes will be connected locally
- **rewiring** (*float*) – the probability of rewiring each edge
- **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.
- **n_connections** (*int or None*) – if specified, the number of efferent synaptic connections per neuron
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or None to make one when needed.
- **safe** (*bool*) – If True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

  Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

create_synaptic_block(*pre_slices, post_slices, pre_vertex_slice, post_vertex_slice, synapse_type, synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

set_projection_information (*machine_time_step*, *synapse_info*)
sets a connectors projection info :param int machine_time_step: machine time step :param SynapseInformation synapse_info: the synapse info

Module contents

class spynnaker.pyNN.models.neural_projections.**DelayAfferentApplicationEdge** (*pre_vertex*, *delay_vertex*, *label=None*)

Bases: pacman.model.graphs.application.application_edge.ApplicationEdge, pacman.model.partitioner_interfaces.abstract_slices_connect.AbstractSlicesConnect

Parameters

- **pre_vertex** (AbstractPopulationVertex) –
- **delay_vertex** (DelayExtensionVertex) –
- **label** (*str*) –

could_connect (*pre_slice*, *post_slice*)
Determine if there is a chance that one of the indexes in the pre-slice could connect to at least one of the indexes in the post-slice.

Note: This method should never return a false negative, but may return a false positives

Parameters

- **pre_slice** (*Slice*) –
- **post_slice** (*Slice*) –

Returns True if a connection could be possible

Return type bool

class spynnaker.pyNN.models.neural_projections.**DelayedApplicationEdge** (*pre_vertex*, *post_vertex*, *synapse_information*, *undelayed_edge*, *label=None*)

Bases: pacman.model.graphs.application.application_edge.ApplicationEdge, pacman.model.partitioner_interfaces.abstract_slices_connect.AbstractSlicesConnect

Parameters

- **pre_vertex** (DelayExtensionVertex) – The delay extension at the start of the edge
- **post_vertex** (AbstractPopulationVertex) – The target of the synapses
- **synapse_information** (SynapseInformation *or* *iterable*(SynapseInformation)) – The synapse information on this edge
- **undelayed_edge** (ProjectionApplicationEdge) – The edge that is used for projections without extended delays
- **label** (*str*) – The edge label

add_synapse_information (*synapse_information*)

Parameters **synapse_information** (SynapseInformation) –

could_connect (*pre_slice*, *post_slice*)

Determine if there is a chance that one of the indexes in the pre-slice could connect to at least one of the indexes in the post-slice.

Note: This method should never return a false negative, but may return a false positives

Parameters

- **pre_slice** (*Slice*) –
- **post_slice** (*Slice*) –

Returns True if a connection could be possible

Return type bool

get_machine_edge (*pre_vertex*, *post_vertex*)

Get a specific machine edge from this edge

Parameters

- **pre_vertex** (DelayExtensionMachineVertex) – The vertex at the start of the machine edge
- **post_vertex** (PopulationMachineVertex) – The vertex at the end of the machine edge

Return type MachineEdge or None

remember_associated_machine_edge (*machine_edge*)

Adds the Machine Edge to the iterable returned by machine_edges

Parameters **machine_edge** (*MachineEdge*) – A pointer to a machine_edge. This edge may not be fully initialised

synapse_information

Return type list(*SynapseInformation*)

undelayed_edge

Get the edge that for projections without extended delays

Return type *ProjectionApplicationEdge*

class spynnaker.pyNN.models.neural_projections.**ProjectionApplicationEdge**(*pre_vertex*, *post_vertex*, *synapse_information*, *label=None*)

Bases: pacman.model.graphs.application.application_edge.ApplicationEdge, pacman.model.partitioner_interfaces.abstract_slices_connect.AbstractSlicesConnect, spinn_front_end_common.interface.provenance.abstract_provides_local_provenance_data.AbstractProvidesLocalProvenanceData

An edge which terminates on an AbstractPopulationVertex.

Parameters

- **pre_vertex** (AbstractPopulationVertex) –
- **post_vertex** (AbstractPopulationVertex) –
- **synapse_information** (SynapseInformation *or iterable*(SynapseInformation)) – The synapse information on this edge
- **label** (*str*) –

add_synapse_information(*synapse_information*)

Parameters **synapse_information** (SynapseInformation) –

could_connect(*pre_slice*, *post_slice*)

Determine if there is a chance that one of the indexes in the pre-slice could connect to at least one of the indexes in the post-slice.

Note: This method should never return a false negative, but may return a false positives

Parameters

- **pre_slice** (*Slice*) –
- **post_slice** (*Slice*) –

Returns True if a connection could be possible

Return type bool

delay_edge

Settable.

Return type *DelayedApplicationEdge* or None

forget_machine_edges()

Clear the collection of machine edges created by this application edge.

get_local_provenance_data()

Get an iterable of provenance data items.

Returns the provenance items

Return type iterable(ProvenanceDataItem)

get_machine_edge(*pre_vertex*, *post_vertex*)

Get a specific machine edge of this edge

Parameters

- **pre_vertex** (PopulationMachineVertex) – The vertex at the start of the machine edge
- **post_vertex** (PopulationMachineVertex) – The vertex at the end of the machine edge

Return type MachineEdge or None

n_delay_stages

Return type int

post_slices
Get the slices for the post_vertexes of the MachineEdges

While the remember machine_edges remain unchanged this will return a list with a consistent id. If the edges change a new list is created

The List will be sorted by lo_atom. No checking is done for overlaps or gaps

Returns Ordered list of post-slices

Return type list(Slice)

pre_slices
Get the slices for the pre_vertexes of the MachineEdges

While the remember machine_edges remain unchanged this will return a list with a consistent id. If the edges change a new list is created

The List will be sorted by lo_atom. No checking is done for overlaps or gaps

Returns Ordered list of pre-slices

Return type list(Slice)

remember_associated_machine_edge(*machine_edge*)
Adds the Machine Edge to the iterable returned by machine_edges

Parameters **machine_edge** (*MachineEdge*) – A pointer to a machine_edge. This edge may not be fully initialised

synapse_information

Return type list(*SynapseInformation*)

class spynnaker.pyNN.models.neural_projections.**SynapseInformation**(*connector*, *pre_population*, *post_population*, *prepop_is_view*, *postpop_is_view*, *rng*, *synapse_dynamics*, *synapse_type*, *is_virtual_machine*, *weights=None*, *delays=None*)

Bases: object

Contains the synapse information including the connector, synapse type and synapse dynamics

Parameters

- **connector** (AbstractConnector) – The connector connected to the synapse
- **pre_population** (Population or PopulationView) – The population sending spikes to the synapse
- **post_population** (Population or PopulationView) – The population hosting the synapse
- **prepop_is_view** (*bool*) – Whether the `pre_population` is a view
- **postpop_is_view** (*bool*) – Whether the `post_population` is a view
- **rng** (*NumpyRNG or None*) – Seeded random number generator
- **synapse_dynamics** (AbstractSynapseDynamics) – The dynamic behaviour of the synapse
- **synapse_type** (AbstractSynapseType) – The type of the synapse
- **is_virtual_machine** (*bool*) – Whether the machine is virtual
- **weights** (*float or list(float) or ndarray(float) or None*) – The synaptic weights
- **delays** (*float or list(float) or ndarray(float) or None*) – The total synaptic delays

add_pre_run_connection_holder(*pre_run_connection_holder*)
Add a connection holder that will be filled in before run

Parameters **pre_run_connection_holder** (ConnectionHolder) – The connection holder to be added

connector
The connector connected to the synapse

Return type *AbstractConnector*

delays
The total synaptic delays (if any)

Return type float or list(float) or ndarray(float) or None

finish_connection_holders()
Finish all the connection holders, and clear the list so that they are not generated again later

may_generate_on_machine()
Do we describe a collection of synapses whose synaptic matrix may be generated on SpiNNaker instead of needing to be calculated in this process and uploaded? This depends on the connector, the definitions of the weights and delays, and the dynamics of the synapses.

Returns True if the synaptic matrix may be generated on machine (or may have already been so done)

Return type bool

n_post_neurons
The number of neurons in the postpopulation

Return type int

n_pre_neurons
The number of neurons in the prepopulation

Return type int

post_population
The population hosting the synapse

Return type *Population* or *PopulationView*

postpop_is_view
Whether the `post_population()` is a view

Return type bool

pre_population
The population sending spikes to the synapse

Return type *Population* or *PopulationView*

pre_run_connection_holders
The list of connection holders to be filled in before run

Return type list(*ConnectionHolder*)

prepop_is_view
Whether the `pre_population()` is a view

Return type bool

rng
Random number generator

Return type NumpyRNG

synapse_dynamics
The dynamic behaviour of the synapse

Return type *AbstractSynapseDynamics*

synapse_type
The type of the synapse

Return type *AbstractSynapseType*

weights
The synaptic weights (if any)

Return type float or list(float) or ndarray(float) or None

spynnaker.pyNN.models.neural_properties package

Module contents

class spynnaker.pyNN.models.neural_properties.**NeuronParameter**(*value*, *data_type*)
Bases: `object`

A settable parameter of a neuron model.

Parameters

- **value** (*int or float or bool or list(int) or list(float) or list(bool) or ndarray or AbstractList*) – what the value of the parameter is; if a list or array, potentially provides a different value for each neuron
- **data_type** (*DataType*) – The serialization type of the parameter in the neuron model.

get_dataspec_datatype()
Get the serialization type of the parameter in the neuron model.

Return type DataType

get_value()

What the value of the parameter is; if a list or array, potentially provides a different value for each neuron.

Return type int or float or bool or list(int) or list(float) or list(bool) or ndarray or AbstractList

iterator_by_slice(*slice_start*, *slice_stop*, *spec*)

Creates an iterator over the commands to use to write the parameter to the data specification being generated.

Parameters

- **slice_start** (*int*) – Inclusive start of the range
- **slice_stop** (*int*) – Exclusive end of the range
- **spec** (*DataSpecificationGenerator*) – The data specification to eventually write to. (Note that this does not actually do the write).

Returns Iterator that produces a command to write to the specification for each element in the slice.

Return type iterator(tuple(bytearray, str))

spynnaker.pyNN.models.neuron package

Subpackages

spynnaker.pyNN.models.neuron.additional_inputs package

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** (*list* (*DataType*)) – 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

Parameters

- **tau_ca2** (*float*, *iterable*(*float*), *RandomDistribution or (mapping) function*) – $\tau_{\mathrm{Ca}^{+2}}$
- **i_ca2** (*float*, *iterable*(*float*), *RandomDistribution or (mapping) function*) – $I_{\mathrm{Ca}^{+2}}$
- **i_alpha** (*float*, *iterable*(*float*), *RandomDistribution or (mapping) function*) – I_α

add_parameters(*parameters*)

Add the initial values of the parameters to the parameter holder

Parameters **parameters** (*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** (*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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*int*) – The time to be advanced in one call to the update of this component
- **ts** – machine time step

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

Return type list(int or float or list(int) or list(float) 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
Settable model parameter: I_{α}

Return type float

i_ca2
Settable model parameter: $I_{\mathrm{Ca}^{+2}}$

Return type float

tau_ca2
Settable model parameter: $\tau_{\mathrm{Ca}^{+2}}$

Return type float

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** (*list (list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update

- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

spynnaker.pyNN.models.neuron.builds package

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.)

> **Warning:** Not currently supported by the tool chain.

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.

> **Warning:** Not currently supported by the tool chain.

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.

> **Warning:** Not currently supported by the tool chain.

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.

Parameters

- **tau_m** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_m
- **cm** (*float, iterable(float), RandomDistribution or (mapping) function*) – C_m
- **v_rest** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{rest}

- **v_reset** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{reset}
- **v_thresh** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{thresh}
- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_e
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_i
- **tau_refrac** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{refrac}
- **i_offset** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_{offset}
- **e_rev_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – E^{rev}_e
- **e_rev_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – E^{rev}_i
- **v** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{init}
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_e
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_i

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.

Parameters

- **tau_m** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_m
- **cm** (*float, iterable(float), RandomDistribution or (mapping) function*) – C_m
- **v_rest** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{rest}
- **v_reset** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{reset}
- **v_thresh** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{thresh}
- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_e
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_i
- **tau_refrac** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{refrac}

- **i_offset** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – I_{offset}
- **v** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – V_{init}
- **exc_response** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – $response_e^{\mathrm{linear}}$
- **exc_exp_response** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – $response_e^{\mathrm{exponential}}$
- **inh_response** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – $response_i^{\mathrm{linear}}$
- **inh_exp_response** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – $response_i^{\mathrm{exponential}}$

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.

Parameters

- **tau_m** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – τ_m
- **cm** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – C_m
- **v_rest** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – V_{rest}
- **v_reset** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – V_{reset}
- **v_thresh** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – V_{thresh}
- **tau_syn_E** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – $\tau^{syn}_{e_1}$
- **tau_syn_E2** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – $\tau^{syn}_{e_2}$
- **tau_syn_I** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – τ^{syn}_i
- **tau_refrac** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – τ_{refrac}
- **i_offset** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – I_{offset}
- **v** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – V_{init}
- **isyn_exc** (*float*, *iterable(float)*, *RandomDistribution or (mapping) function*) – $I^{syn}_{e_1}$

- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_i
- **isyn_exc2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $I^{syn}_{e_2}$

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.

Parameters

- **tau_m** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_m
- **cm** (*float, iterable(float), RandomDistribution or (mapping) function*) – C_m
- **v_rest** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{rest}
- **v_reset** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{reset}
- **v_thresh** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{thresh}
- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_e
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_i
- **tau_refrac** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{refrac}
- **i_offset** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_{offset}
- **v** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{init}
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_e
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_i

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

Warning: Not currently supported by the tool chain.

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

Izhikevich neuron model with conductance inputs.

Parameters

- **a** (*float, iterable(float), RandomDistribution or (mapping) function*) – a
- **b** (*float, iterable(float), RandomDistribution or (mapping) function*) – b
- **c** (*float, iterable(float), RandomDistribution or (mapping) function*) – c
- **d** (*float, iterable(float), RandomDistribution or (mapping) function*) – d
- **i_offset** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_{offset}
- **u** (*float, iterable(float), RandomDistribution or (mapping) function*) – $u_{init} = \delta V_{init}$
- **v** (*float, iterable(float), RandomDistribution or (mapping) function*) – $v_{init} = V_{init}$
- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_e
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_i
- **e_rev_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – E^{rev}_e
- **e_rev_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – E^{rev}_i
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_e
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_i

class spynnaker.pyNN.models.neuron.builds.**IzkCurrExpBase**(***kwargs*)
Bases: spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard

Izhikevich neuron model with current inputs.

Parameters

- **a** (*float, iterable(float), RandomDistribution or (mapping) function*) – a
- **b** (*float, iterable(float), RandomDistribution or (mapping) function*) – b
- **c** (*float, iterable(float), RandomDistribution or (mapping) function*) – c

- **d** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – d
- **i_offset** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – I_{offset}
- **u** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – $u_{init} = \delta V_{init}$
- **v** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – $v_{init} = V_{init}$
- **tau_syn_E** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – τ^{syn}_e
- **tau_syn_I** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – τ^{syn}_i
- **isyn_exc** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – I^{syn}_e
- **isyn_inh** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – I^{syn}_i

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.

Habenschuss S, Jonke Z, Maass W. Stochastic computations in cortical microcircuit models. *PLoS Computational Biology.* 2013;9(11):e1003311. doi:10.1371/journal.pcbi.1003311

Parameters

- **tau_m** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – τ_m
- **cm** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – C_m
- **v_rest** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – V_{rest}
- **v_reset** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – V_{reset}
- **v_thresh** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – V_{thresh}
- **tau_syn_E** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – τ^{syn}_e
- **tau_syn_I** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – τ^{syn}_i
- **tau_refrac** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – τ_{refrac}
- **i_offset** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – I_{offset}
- **e_rev_E** (*float*, *iterable(float)*, *RandomDistribution* or *(mapping) function*) – E^{rev}_e

- **e_rev_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – E_i^{rev}
- **du_th** (*float, iterable(float), RandomDistribution or (mapping) function*) – du_{thresh}
- **tau_th** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{thresh}
- **v** (*Float, float, iterable(float), RandomDistribution or (mapping) function*) – V_{init}
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_e^{syn}
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_i^{syn}

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.

Parameters

- **tau_m** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_m
- **cm** (*float, iterable(float), RandomDistribution or (mapping) function*) – C_m
- **v_rest** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{rest}
- **v_reset** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{reset}
- **v_thresh** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{thresh}
- **tau_refrac** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{refrac}
- **i_offset** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_{offset}
- **v** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{init}
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_e^{syn}
- **isyn_inh** – I_i^{syn}

Type isyn_inh: float, iterable(float), RandomDistribution or (mapping) function

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

Parameters

- **tau_m** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_m
- **cm** (*float, iterable(float), RandomDistribution or (mapping) function*) – C_m
- **v_rest** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{rest}
- **v_reset** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{reset}
- **v_thresh** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{thresh}
- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_e
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_i
- **tau_refrac** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{refrac}
- **i_offset** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_{offset}
- **tau_ca2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $\tau_{\mathrm{Ca}^{+2}}$
- **i_ca2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $I_{\mathrm{Ca}^{+2}}$
- **i_alpha** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_α
- **v** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{init}
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_e
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_i

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)

Parameters

- **tau_m** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_m
- **cm** (*float, iterable(float), RandomDistribution or (mapping) function*) – C_m
- **v_rest** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{rest}

- **v_reset** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{reset}
- **v_thresh** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{thresh}
- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – $\tau^{syn}_{e_1}$
- **tau_syn_E2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $\tau^{syn}_{e_2}$
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_{i}
- **tau_refrac** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{refrac}
- **i_offset** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_{offset}
- **v** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{init}
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – $I^{syn}_{e_1}$
- **isyn_exc2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $I^{syn}_{e_2}$
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_{i}
- **multiplicator** (*float, iterable(float), RandomDistribution or (mapping) function*) –
- **exc2_old** (*float, iterable(float), RandomDistribution or (mapping) function*) –
- **scaling_factor** (*float, iterable(float), RandomDistribution or (mapping) function*) –

spynnaker.pyNN.models.neuron.implementations package

Module contents

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** (*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** (*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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of the vertex to generate parameters for

Return type ndarray(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_data_types()
Get the data type of the variables that can be recorded

Returns dict of name of variable to DataType of variable

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(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 list(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

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* (read from the machine)

Parameters

- **data** (*bytearray or bytes or memoryview*) – The data to be read
- **offset** (*int*) – The offset where the data should be read from
- **vertex_slice** (*Slice*) – The slice of the vertex to read parameters for
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

class spynnaker.pyNN.models.neuron.implementations.**AbstractStandardNeuronComponent**(*data_types*
Bases: object

Represents a component of a standard neural model.

Parameters **data_types** (*list (DataType)*) – 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** (*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** (*RangeDictionary*) – A holder of the state variables

get_data (*parameters*, *state_variables*, *vertex_slice*, *ts*)
Get the data *to be written to the machine* for this model.

Parameters

- **parameters** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of the vertex to generate parameters for

Return type ndarray(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*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*float*) – The time to be advanced in one call to the update of this component

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

Return type list(int or float or list(int) or list(float) 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* (read from the machine)

Parameters

- **data** (*bytes or bytearray*) – The data to be read
- **offset** (*int*) – The offset where the data should be read from
- **vertex_slice** (*Slice*) – The slice of the vertex to read parameters for
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

Returns The offset after reading the data

Return type int

struct

The structure of the component. This structure will have copies in both SDRAM (the initialisation values) and DTCM (the working copy).

Return type *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** (*list (list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

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 componentised neuron implementation.

Parameters

- **model_name** (*str*) –
- **binary** (*str*) –
- **neuron_model** (AbstractNeuronModel) –
- **input_type** (AbstractInputType) –
- **synapse_type** (AbstractSynapseType) –
- **threshold_type** (AbstractThresholdType) –
- **additional_input_type** (AbstractAdditionalInput *or None*) –

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

Parameters **parameters** (*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** (*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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of the vertex to generate parameters for

Return type ndarray(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_data_types ()
Get the data type of the variables that can be recorded

Returns dict of name of variable to DataType of variable

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(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 list(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

Return type bool

model_name
The name of the model

Return type str

n_steps_per_timestep

read_data(*data*, *offset*, *vertex_slice*, *parameters*, *state_variables*)
Read the parameters and state variables of the model *from the given data* (read from the machine)

Parameters

- **data** (*bytearray or bytes or memoryview*) – The data to be read
- **offset** (*int*) – The offset where the data should be read from
- **vertex_slice** (*Slice*) – The slice of the vertex to read parameters for
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*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

Parameters

- **ranged_dict** (*RangeDictionary*) –
- **vertex_slice** (*Slice*) –

spynnaker.pyNN.models.neuron.input_types package

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** (*list (DataType)*) – 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

Parameters

- **e_rev_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – Reversal potential for excitatory input; E^{rev}_e
- **e_rev_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – Reversal potential for inhibitory input; E^{rev}_i

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

Parameters **parameters** (*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** (*RangeDictionary*) – A holder of the state variables

e_rev_E
E_{rev_e}

e_rev_I
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*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*float*) – The time to be advanced in one call to the update of this component

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

Return type list(int or float or list(int) or list(float) 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** (*list(list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – 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** (*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** (*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*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*float*) – The time to be advanced in one call to the update of this component

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

Return type list(int or float or list(int) or list(float) 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** (*list(list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – 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.

Parameters

- **multiplicator** (*float, iterable(float), RandomDistribution or (mapping) function*) –
- **inh_input_previous** (*float, iterable(float), RandomDistribution or (mapping) function*) –

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

Parameters **parameters** (*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** (*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*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*float*) – The time to be advanced in one call to the update of this component

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

Return type list(int or float or list(int) or list(float) 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** (*list(list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

class spynnaker.pyNN.models.neuron.input_types.**InputTypeDelta**

Bases: spynnaker.pyNN.models.neuron.input_types.abstract_input_type.AbstractInputType

The delta input type

add_parameters(*parameters*)

Add the initial values of the parameters to the parameter holder

Parameters **parameters** (*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** (*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*, *ts*)

Get the values to be written to the machine for this model

Parameters

- **parameters** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*float*) – The time to be advanced in one call to the update of this component

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

Return type list(int or float or list(int) or list(float) 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** (*list (list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update

- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

spynnaker.pyNN.models.neuron.neuron_models package

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** (*list(DataType)*) – A list of data types in the neuron structure, in the order that they appear
- **global_data_types** (*list(DataType) or None*) – A list of data types in the neuron global structure, in the order that they appear

get_data(*parameters*, *state_variables*, *vertex_slice*, *ts*)

Get the data *to be written to the machine* for this model.

Parameters

- **parameters** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of the vertex to generate parameters for

Return type ndarray(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(*ts*)

Get the global values to be written to the machine for this model

Parameters **ts** (*float*) – The time to advance the model at each call

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

Return type list(int or float) or ndarray

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

Return type *Struct*

read_data (*data*, *offset*, *vertex_slice*, *parameters*, *state_variables*)
Read the parameters and state variables of the model *from the given data* (read from the machine)

Parameters

- **data** (*bytes or bytearray*) – The data to be read
- **offset** (*int*) – The offset where the data should be read from
- **vertex_slice** (*Slice*) – The slice of the vertex to read parameters for
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

Returns The offset after reading the data

Return type int

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

Model of neuron due to Eugene M. Izhikevich et al

Parameters

- **a** (*float, iterable(float), RandomDistribution or (mapping) function*) – a
- **b** (*float, iterable(float), RandomDistribution or (mapping) function*) – b
- **c** (*float, iterable(float), RandomDistribution or (mapping) function*) – c
- **d** (*float, iterable(float), RandomDistribution or (mapping) function*) – d
- **v_init** (*float, iterable(float), RandomDistribution or (mapping) function*) – v_{init}
- **u_init** (*float, iterable(float), RandomDistribution or (mapping) function*) – u_{init}
- **i_offset** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_{offset}

a
Settable model parameter: a

Return type float

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

Parameters **parameters** (*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** (*RangeDictionary*) – A holder of the state variables

b
Settable model parameter: b

Return type float

c
Settable model parameter: c

Return type float

d
Settable model parameter: d

Return type float

get_global_values (*ts*)
Get the global values to be written to the machine for this model

Parameters **ts** (*float*) – The time to advance the model at each call

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

Return type list(int or float) or ndarray

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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*float*) – The time to be advanced in one call to the update of this component
- **ts** – machine time step

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

Return type list(int or float or list(int) or list(float) 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
Settable model parameter: I_{offset}

Return type float

u_init
Settable model parameter: u_{init}

Return type float

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

v_init
Settable model parameter: v_{init}

Return type float

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

Classic leaky integrate and fire neuron model.

Parameters

- **v_init** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{init}
- **v_rest** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{rest}
- **tau_m** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_m
- **cm** (*float, iterable(float), RandomDistribution or (mapping) function*) – C_m
- **i_offset** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_{offset}
- **v_reset** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{reset}
- **tau_refrac** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{refrac}

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

Parameters **parameters** (*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** (*RangeDictionary*) – A holder of the state variables

cm
Settable model parameter: C_m

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*, *ts*)
Get the values to be written to the machine for this model

Parameters

- **parameters** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*int*) – The time to be advanced in one call to the update of this component
- **ts** – machine time step

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

Return type list(int or float or list(int) or list(float) 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
Settable model parameter: I_{offset}

Return type float

tau_m
Settable model parameter: τ_m

Return type float

tau_refrac
Settable model parameter: τ_{refrac}

Return type float

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** (*list (list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update

- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

v_init
Settable model parameter: V_{init}

Return type float

v_reset
Settable model parameter: V_{reset}

Return type float

v_rest
Settable model parameter: V_{rest}

Return type float

spynnaker.pyNN.models.neuron.plasticity package

Subpackages

spynnaker.pyNN.models.neuron.plasticity.stdp package

Subpackages

spynnaker.pyNN.models.neuron.plasticity.stdp.synapse_structure package

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

Return type int

get_weight_half_word()
The index of the half-word where the weight should be written

Return type int

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

Return type int

get_weight_half_word()
The index of the half-word where the weight should be written

Return type int

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

Return type int

get_weight_half_word()

The index of the half-word where the weight should be written

Return type int

spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence package

Module contents

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**AbstractTimingDependence**

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

Return type int

get_provenance_data(*pre_population_label*, *post_population_label*)

Get any provenance data

Parameters

- **pre_population_label** (*str*) – label of pre.
- **post_population_label** (*str*) – label of post.

Return type list(ProvenanceDataItem)

is_same_as(*timing_dependence*)

Determine if this timing dependence is the same as another

Parameters **timing_dependence** (AbstractTimingDependence) –

Return type bool

n_weight_terms

The number of weight terms expected by this timing rule

Return type int

pre_trace_n_bytes

The number of bytes used by the pre-trace of the rule per neuron

Return type int

synaptic_structure

Get the synaptic structure of the plastic part of the rows

Return type *AbstractSynapseStructure*

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters (*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **machine_time_step** (*int*) –
- **weight_scales** (*dict* (SynapseInformation, *float*)) – (unused?)

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**TimingDependenceSpikeI**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence

A basic timing dependence STDP rule.

Parameters

- **tau_plus** (*float*) – τ_+
- **tau_minus** (*float*) – τ_-
- **A_plus** (*float*) – A^+
- **A_minus** (*float*) – A^-

A_minus
A^-

Return type float

A_plus
A^+

Return type float

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

Return type int

is_same_as (*timing_dependence*)
Determine if this timing dependence is the same as another

Parameters **timing_dependence** (AbstractTimingDependence) –

Return type bool

n_weight_terms
The number of weight terms expected by this timing rule

Return type int

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

Return type int

synaptic_structure
Get the synaptic structure of the plastic part of the rows

Return type *AbstractSynapseStructure*

tau_minus
τ_-

Return type float

tau_plus
τ_+

Return type float

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **machine_time_step** (*int*) –
- **weight_scales** (*dict(SynapseInformation, float)*) – (unused?)

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**TimingDependencePfiste**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence

A timing dependence STDP rule based on spike triplets.

Jean-Pascal Pfister, Wulfram Gerstner. Triplets of Spikes in a Model of Spike Timing-Dependent Plasticity. *Journal of Neuroscience*, 20 September 2006, 26 (38) 9673-9682; DOI: 10.1523/JNEUROSCI.1425-06.2006

Parameters

- **tau_plus** (*float*) – τ_+
- **tau_minus** (*float*) – τ_-
- **tau_x** (*float*) – τ_x
- **tau_y** (*float*) – τ_y
- **A_plus** (*float*) – A^+
- **A_minus** (*float*) – A^-

A_minus
A^-

Return type float

A_plus
A^+

Return type float

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

Return type int

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

Parameters **timing_dependence** (AbstractTimingDependence) –

Return type bool

n_weight_terms
The number of weight terms expected by this timing rule

Return type int

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

Return type int

synaptic_structure
Get the synaptic structure of the plastic part of the rows

Return type *AbstractSynapseStructure*

tau_minus
τ_-

Return type float

tau_plus
τ_+

Return type float

tau_x
τ_x

Return type float

tau_y
τ_y

Return type float

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **machine_time_step** (*int*) –
- **weight_scales** (*dict* (SynapseInformation, *float*)) – (unused?)

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**TimingDependenceRecurr**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence

A timing dependence STDP rule based on recurrences.

Parameters

- **accumulator_depression** (*int*) –
- **accumulator_potentiation** (*int*) –
- **mean_pre_window** (*float*) –
- **mean_post_window** (*float*) –
- **dual_fsm** (*bool*) –
- **A_plus** (*float*) – A^+
- **A_minus** (*float*) – A^-

A_minus
A^-

Return type float

A_plus
A^+

Return type float

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

Return type int

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

Parameters **timing_dependence** (AbstractTimingDependence) –

Return type bool

n_weight_terms
The number of weight terms expected by this timing rule

Return type int

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

Return type int

synaptic_structure
Get the synaptic structure of the plastic part of the rows

Return type *AbstractSynapseStructure*

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters (*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **machine_time_step** (*int*) –
- **weight_scales** (*dict (SynapseInformation, float)*) – (unused?)

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**TimingDependenceSpikeN**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence

A timing dependence STDP rule based on nearest pairs.

Parameters

- **tau_plus** (*float*) – τ_+
- **tau_minus** (*float*) – τ_-
- **A_plus** (*float*) – A^+
- **A_minus** (*float*) – A^-

A_minus
A^-

Return type float

A_plus
A^+

Return type float

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

Return type int

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

Parameters **timing_dependence** (AbstractTimingDependence) –

Return type bool

n_weight_terms
The number of weight terms expected by this timing rule

Return type int

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

Return type int

synaptic_structure
Get the synaptic structure of the plastic part of the rows

Return type *AbstractSynapseStructure*

tau_minus
τ_-

Return type float

tau_plus
τ_+

Return type float

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **machine_time_step** (*int*) –
- **weight_scales** (*dict* (SynapseInformation, *float*)) – (unused?)

class spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.**TimingDependenceVogels**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.abstract_timing_dependence.AbstractTimingDependence

A timing dependence STDP rule due to Vogels (2011).

Parameters

- **alpha** (*float*) – α

- **tau** (*float*) – τ
- **A_plus** (*float*) – A^+
- **A_minus** (*float*) – A^-

A_minus
A^-

Return type float

A_plus
A^+

Return type float

alpha
α

Return type float

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

Return type int

is_same_as(*timing_dependence*)
Determine if this timing dependence is the same as another

Parameters **timing_dependence** (AbstractTimingDependence) –

Return type bool

n_weight_terms
The number of weight terms expected by this timing rule

Return type int

pre_trace_n_bytes
The number of bytes used by the pre-trace of the rule per neuron

Return type int

synaptic_structure
Get the synaptic structure of the plastic part of the rows

Return type *AbstractSynapseStructure*

tau
τ

Return type float

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*, *machine_time_step*, *weight_scales*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **machine_time_step** (*int*) –
- **weight_scales** (*dict (SynapseInformation, float)*) – (unused?)

spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence package

Module contents

class spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.**AbstractHasAPlusAMinus**

Bases: object

An object that has A^+ and A^- properties.

A_minus

Settable model parameter: A^-

Return type float

A_plus

Settable model parameter: A^+

Return type float

set_a_plus_a_minus (*a_plus*, *a_minus*)

Set the values of A^+ and A^-.

Parameters

- **a_plus** (*float*) – A^+
- **a_minus** (*float*) – A^-

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

Parameters

- **n_synapse_types** (*int*) –
- **n_weight_terms** (*int*) –

Return type int

get_provenance_data (*pre_population_label*, *post_population_label*)

Get any provenance data

Parameters

- **pre_population_label** (*str*) – label of pre.
- **post_population_label** (*str*) – label of post.

Returns the provenance data of the weight dependency

Return type list(ProvenanceDataItem)

is_same_as (*weight_dependence*)
Determine if this weight dependence is the same as another

Parameters **weight_dependence** (AbstractWeightDependence) –

Return type bool

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

Return type float

write_parameters (*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **machine_time_step** (*int*) – (unused?)
- **weight_scales** (*iterable(float)*) –
- **n_weight_terms** (*int*) –

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

An additive weight dependence STDP rule.

Parameters

- **w_min** (*float*) – w^{min}
- **w_max** (*float*) – w^{max}

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

Parameters

- **n_synapse_types** (*int*) –
- **n_weight_terms** (*int*) –

Return type int

is_same_as (*weight_dependence*)
Determine if this weight dependence is the same as another

Parameters **weight_dependence** (AbstractWeightDependence) –

Return type bool

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

w_max
w^{max}

Return type float

w_min
w^{min}

Return type float

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

Return type float

write_parameters(*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **machine_time_step** (*int*) – (unused?)
- **weight_scales** (*iterable(float)*) –
- **n_weight_terms** (*int*) –

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

A multiplicative weight dependence STDP rule.

Parameters

- **w_min** (*float*) – w^{min}
- **w_max** (*float*) – w^{max}

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

Parameters

- **n_synapse_types** (*int*) –
- **n_weight_terms** (*int*) –

Return type int

is_same_as(*weight_dependence*)
Determine if this weight dependence is the same as another

Parameters **weight_dependence** (AbstractWeightDependence) –

Return type bool

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

w_max
w^{max}

Return type float

w_min
w^{min}

Return type float

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

Return type float

write_parameters (*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **machine_time_step** (*int*) – (unused?)
- **weight_scales** (*iterable(float)*) –
- **n_weight_terms** (*int*) –

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

An triplet-based additive weight dependence STDP rule.

Parameters

- **w_min** (*float*) – w^{min}
- **w_max** (*float*) – w^{max}
- **A3_plus** (*float*) – A_3^+
- **A3_minus** (*float*) – A_3^-

A3_minus
A_3^-

Return type float

A3_plus
A_3^+

Return type float

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

Parameters

- **n_synapse_types** (*int*) –
- **n_weight_terms** (*int*) –

Return type int

is_same_as(*weight_dependence*)
Determine if this weight dependence is the same as another

Parameters **weight_dependence** (AbstractWeightDependence) –

Return type bool

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

w_max
w^{max}

Return type float

w_min
w^{min}

Return type float

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

Return type float

write_parameters(*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **machine_time_step** (*int*) – (unused?)
- **weight_scales** (*iterable(float)*) –
- **n_weight_terms** (*int*) –

Submodules

spynnaker.pyNN.models.neuron.plasticity.stdp.common module

spynnaker.pyNN.models.neuron.plasticity.stdp.common.**float_to_fixed**(*value*)

Parameters **value** (*float*) –

Return type int

`spynnaker.pyNN.models.neuron.plasticity.stdp.common.`**`get_exp_lut_array`**(*time_step*, *time_constant*, *shift=0*)

Parameters

- **time_step** (*int*) –
- **time_constant** (*float*) –
- **shift** (*int*) –

Return type ndarray

Module contents

Module contents

spynnaker.pyNN.models.neuron.structural_plasticity package

Subpackages

spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis package

Subpackages

spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.elimination package

Module contents

class `spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.elimination.`**`Abstrac`**

Bases: `object`

An elimination rule

`get_parameter_names`()

Return the names of the parameters supported by this rule

Return type iterable(str)

`get_parameters_sdram_usage_in_bytes`()

Get the amount of SDRAM used by the parameters of this rule

Return type int

`vertex_executable_suffix`

The suffix to be appended to the vertex executable for this rule

Return type str

`write_parameters`(*spec*, *weight_scale*)

Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –

- **weight_scale** (*float*) –

class spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.elimination.**RandomB**

Bases: spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.elimination.abstract_elimination.AbstractElimination

Elimination Rule that depends on the weight of a synapse

Parameters

- **threshold** (*float*) – Below this weight is considered depression, above or equal to this weight is considered potentiation (or the static weight of the connection on static weight connections)
- **prob_elim_depressed** (*float*) – The probability of elimination if the weight has been depressed (ignored on static weight connections)
- **prob_elim_potentiated** (*float*) – The probability of elimination of the weight has been potentiated or has not changed (and also used on static weight connections)

get_parameter_names()
Return the names of the parameters supported by this rule

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

Return type int

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*, *weight_scale*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **weight_scale** (*float*) –

spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.formation package

Module contents

class spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.formation.**AbstractF**
Bases: object

A formation rule

get_parameter_names()
Return the names of the parameters supported by this rule

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

Return type int

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters (*spec*)
Write the parameters of the rule to the spec

Parameters **spec** (*DataSpecificationGenerator*) –

class spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.formation.**DistanceI**

Bases: spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.formation.abstract_formation.AbstractFormation

Formation rule that depends on the physical distance between neurons

Parameters

- **grid** (*tuple(int,int) or list(int) or ndarray(int)*) – (x, y) dimensions of the grid of distance
- **p_form_forward** (*float*) – The peak probability of formation on feed-forward connections
- **sigma_form_forward** (*float*) – The spread of probability with distance of formation on feed-forward connections
- **p_form_lateral** (*float*) – The peak probability of formation on lateral connections
- **sigma_form_lateral** (*float*) – The spread of probability with distance of formation on lateral connections

distance (*x0, x1, metric*)
Compute the distance between points x0 and x1 place on the grid using periodic boundary conditions.

Parameters

- **x0** (*ndarray(int)*) – first point in space
- **x1** (*ndarray(int)*) – second point in space
- **grid** (*ndarray(int)*) – shape of grid
- **metric** (*str*) – distance metric, i.e. euclidian or manhattan or equidistant

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 ndarray(float)

get_parameter_names()
Return the names of the parameters supported by this rule

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

Return type int

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*)
Write the parameters of the rule to the spec

Parameters **spec** (*DataSpecificationGenerator*) –

spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.partner_selection package

Module contents

class spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.partner_selection.A
Bases: object

A partner selection rule

get_parameter_names()
Return the names of the parameters supported by this rule

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

Return type str

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*)
Write the parameters of the rule to the spec

Parameters **spec** (*DataSpecificationGenerator*) –

class spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.partner_selection.I
Bases: spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.partner_selection.abstract_partner_selection.AbstractPartnerSelection

Partner selection that picks a random source neuron from the neurons that spiked in the last timestep

Parameters **spike_buffer_size** – The size of the buffer for holding spikes

get_parameter_names()
Return the names of the parameters supported by this rule

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

Return type str

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*)
Write the parameters of the rule to the spec

Parameters **spec** (*DataSpecificationGenerator*) –

class spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.partner_selection.R
Bases: spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.partner_selection.abstract_partner_selection.AbstractPartnerSelection

Partner selection that picks a random source neuron from all sources

get_parameter_names()
Return the names of the parameters supported by this rule

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

Return type str

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*)
Write the parameters of the rule to the spec

Parameters **spec** (*DataSpecificationGenerator*) –

Module contents

Module contents

spynnaker.pyNN.models.neuron.synapse_dynamics package

Module contents

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 ndarray(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.**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.

Parameters **fp_size** (*ndarray*) –

get_n_plastic_plastic_words_per_row(*pp_size*)
Get the number of plastic plastic words to be read from each row.

Parameters **pp_size** (*ndarray*) –

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*.

Parameters

- **pp_size** (*ndarray*) –
- **fp_size** (*ndarray*) –

get_n_words_for_plastic_connections(*n_connections*)
Get the number of 32-bit words for *n_connections* in a single row.

Parameters **n_connections** (*int*) –

Return type int

get_plastic_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*, *max_n_synapses*)
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.

Parameters

- **connections** (*ndarray*) – The connections to get data for
- **connection_row_indices** (*ndarray*) – The row into which each connection should go
- **n_rows** (*int*) – The total number of rows

- **post_vertex_slice** (*Slice*) – The slice of the post vertex to get the connections for
- **n_synapse_types** (*int*) – The number of synapse types
- **max_n_synapses** (*int*) – The maximum number of synapses to generate

Returns (fp_data, pp_data, fp_size, pp_size)

Return type tuple(ndarray, ndarray, ndarray, ndarray)

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*.

Parameters

- **post_vertex_slice** (*Slice*) –
- **n_synapse_types** (*int*) –
- **pp_size** (*ndarray*) – 1D
- **pp_data** (*ndarray*) – 2D
- **fp_size** (*ndarray*) – 1D
- **fp_data** (*ndarray*) – 2D

Returns array with columns `source`, `target`, `weight`, `delay`

Return type ndarray

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.

Parameters **ff_size** (*ndarray*) –

Return type ndarray

get_n_synapses_in_rows (*ff_size*)

Get the number of synapses in the rows with sizes *ff_size*.

Parameters **ff_size** (*ndarray*) –

Return type ndarray

get_n_words_for_static_connections (*n_connections*)

Get the number of 32-bit words for *n_connections* in a single row.

Parameters **n_connections** (*int*) –

Return type int

get_static_synaptic_data (*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*, *max_n_synapses*)

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.

Parameters

- **connections** (*ndarray*) – The connections to get data for
- **connection_row_indices** (*ndarray*) – The row into which each connection should go
- **n_rows** (*int*) – The number of rows to write
- **post_vertex_slice** (*Slice*) – The slice of the post vertex to generate for
- **n_synapse_types** (*int*) – The number of synapse types
- **max_n_synapses** (*int*) – The maximum number of synapses to generate

Returns (ff_data, ff_size)

Return type tuple(list(ndarray), ndarray)

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*.

Parameters

- **post_vertex_slice** (*Slice*) –
- **n_synapse_types** (*int*) –
- **ff_size** (*ndarray*) –
- **ff_data** (*list (ndarray)*) –

class spynnaker.pyNN.models.neuron.synapse_dynamics.**AbstractSynapseDynamics**
Bases: object

How do the dynamics of a synapse interact with the rest of the model.

NUMPY_CONNECTORS_DTYPE = [('source', 'uint32'), ('target', 'uint32'), ('weight', 'floa
Type model of the basic configuration data of a connector

are_weights_signed()
Determines if the weights are signed values

Return type bool

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*, *max_n_synapses*)
Converts per-connection data generated from connections into row-based data to be returned from get_synaptic_data

Parameters

- **connection_row_indices** (*ndarray*) – The index of the row that each item should go into
- **n_rows** (*int*) – The number of rows
- **data** (*ndarray*) – The non-row-based data
- **max_n_synapses** (*int*) – The maximum number of synapses to generate in each row

Return type list(ndarray)

delay
The delay of connections

get_delay_maximum(*connector*, *synapse_info*)
Get the maximum delay for the synapses

Parameters

- **connector** (AbstractConnector) –
- **delays** (*ndarray*) –

get_delay_minimum(*connector*, *synapse_info*)
Get the minimum delay for the synapses. This will support the filtering of the undelayed edge from the graph, but requires fixes in the synaptic manager to happen first before this can be utilised fully.

Parameters

- **connector** (AbstractConnector) – connector
- **synapse_info** (*ndarray*) – synapse info

get_delay_variance(*connector*, *delays*, *synapse_info*)
Get the variance in delay for the synapses

Parameters

- **connector** (AbstractConnector) –
- **delays** (*ndarray*) –

get_max_synapses(*n_words*)
Get the maximum number of synapses that can be held in the given number of words

Parameters **n_words** (*int*) – 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

Parameters

- **rows** (*ndarray*) –
- **item_size** (*int*) –

Return type ndarray

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

Parameters

- **n_neurons** (*int*) –
- **n_synapse_types** (*int*) –

Return type int

get_provenance_data(*pre_population_label*, *post_population_label*)
Get the provenance data from this synapse dynamics object

Parameters

- **pre_population_label** (*str*) –
- **post_population_label** (*str*) –

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

Return type str

get_weight_maximum(*connector*, *synapse_info*)
Get the maximum weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

get_weight_mean(*connector*, *synapse_info*)
Get the mean weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

get_weight_variance(*connector*, *weights*, *synapse_info*)
Get the variance in weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

get_words(*rows*)
Convert the row data to words

Parameters **rows** (*ndarray*) –

Return type ndarray

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

Return type bool

merge(*synapse_dynamics*)
Merge with the given synapse_dynamics and return the result, or error if merge is not possible

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

Return type *AbstractSynapseDynamics*

pad_to_length
The amount each row should pad to, or None if not specified

set_delay(*delay*)
Set the delay

weight
The weight of connections

write_parameters (*spec*, *region*, *machine_time_step*, *weight_scales*)
Write the synapse parameters to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **region** (*int*) – region ID
- **machine_time_step** (*int*) –
- **weight_scales** (*list (float)*) –

class spynnaker.pyNN.models.neuron.synapse_dynamics.**AbstractSynapseDynamicsStructural**
Bases: object

check_initial_delay (*max_delay_ms*)
Check that delays can be done without delay extensions

Parameters **max_delay_ms** (*int*) – The maximum delay supported, in milliseconds

Raises **Exception** – if the delay is out of range

elimination
The elimination rule

Return type *AbstractElimination*

f_rew
The frequency of rewiring

Return type float

formation
The formation rule

Return type *AbstractFormation*

get_structural_parameters_sdram_usage_in_bytes (*graph*, *vertex*, *n_neurons*)
Get the size of the structural parameters

Note: At the Application level this will be an estimate.

Parameters

- **graph** (*ApplicationGraph or MachineGraph*) – Graph at same level as vertex.
- **vertex** (*ApplicationVertex or MachineVertex*) – Vertex at the same level as the graph
- **n_neurons** (*int*) –

Returns the size of the parameters, in bytes

Return type int

Raises **PacmanInvalidParameterException** –

initial_delay
The delay of a formed connection

Return type float or (float, float)

initial_weight
The weight of a formed connection

Return type float

partner_selection
The partner selection rule

Return type *AbstractPartnerSelection*

s_max
The maximum number of synapses

Return type int

seed
The seed to control the randomness

set_connections (*connections*, *post_vertex_slice*, *app_edge*, *synapse_info*, *machine_edge*)
Set connections for structural plasticity

Parameters

- **connections** (*ndarray*) –
- **post_vertex_slice** (*Slice*) –
- **app_edge** (ProjectionApplicationEdge) –
- **synapse_info** (SynapseInformation) –
- **machine_edge** (*MachineEdge*) –

with_replacement
Whether to allow replacement when creating synapses

Return type bool

write_structural_parameters (*spec*, *region*, *machine_time_step*, *weight_scales*, *machine_graph*, *machine_vertex*, *routing_info*, *synaptic_matrices*)
Write structural plasticity parameters

Parameters

- **spec** (*DataSpecificationGenerator*) – The data specification to write to
- **region** (*int*) – region ID
- **machine_time_step** (*float*) – The simulation time step
- **weight_scales** (*list (float)*) – Weight scaling for each synapse type
- **machine_graph** (*MachineGraph*) – The machine graph
- **machine_vertex** (AbstractPopulationVertex) – The machine vertex
- **routing_info** (*RoutingInfo*) – Routing information for all edges
- **synaptic_matrices** (SynapticMatrices) – The synaptic matrices for this vertex

spynnaker.pyNN.models.neuron.synapse_dynamics.**calculate_spike_pair_additive_stdp_weight**(*pre_ pos ini- tial_ plas tic_ max a_p a_m tau_ tau_*

Calculates the expected stdp weight for SpikePair Additive STDP

Parameters

- **pre_spikes** –
- **post_spikes** –
- **initial_weight** –
- **plastic_delay** –
- **max_weight** –
- **a_plus** –
- **a_minus** –
- **tau_plus** –
- **tau_minus** –

Returns

spynnaker.pyNN.models.neuron.synapse_dynamics.**calculate_spike_pair_multiplicative_stdp_weig**

Calculates the expected stdp weight for SpikePair Multiplicative STDP

Parameters

- **pre_spikes** (*iterable(int)*) – Spikes going into the model
- **post_spikes** (*iterable(int)*) – Spikes recorded on the model
- **initial_weight** (*float*) – Starting weight for the model
- **plastic_delay** (*int*) – param of the stdp model
- **min_weight** (*float*) – param of the stdp model
- **max_weight** (*float*) – param of the stdp model
- **a_plus** (*float*) – param of the stdp model

- **a_minus** (*float*) – param of the stdp model
- **tau_plus** (*float*) – param of the stdp model
- **tau_minus** (*float*) – param of the stdp model

Returns

class spynnaker.pyNN.models.neuron.synapse_dynamics.**PyNNSynapseDynamics**(*slow=None*, *fast=None*)

Bases: object

slow

class spynnaker.pyNN.models.neuron.synapse_dynamics.**SynapseDynamicsStatic**(*weight=<sphinx.ext.auto object>*, *delay=None*, *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

The dynamics of a synapse that does not change over time.

Parameters

- **weight** (*float*) –
- **delay** (*float or None*) – Use None to get the simulator default minimum delay.
- **pad_to_length** (*int*) –

are_weights_signed()

Determines if the weights are signed values

Return type bool

changes_during_run

Determine if the synapses change during a run

Return type bool

delay

The delay of connections

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 (*int*) – 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.

Parameters ff_size (*ndarray*) –

Return type ndarray

get_n_synapses_in_rows (*ff_size*)

Get the number of synapses in the rows with sizes *ff_size*.

Parameters **ff_size** (*ndarray*) –

Return type ndarray

get_n_words_for_static_connections (*n_connections*)

Get the number of 32-bit words for *n_connections* in a single row.

Parameters **n_connections** (*int*) –

Return type int

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

Parameters

- **n_neurons** (*int*) –
- **n_synapse_types** (*int*) –

Return type int

get_static_synaptic_data (*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*, *max_n_synapses*)

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.

Parameters

- **connections** (*ndarray*) – The connections to get data for
- **connection_row_indices** (*ndarray*) – The row into which each connection should go
- **n_rows** (*int*) – The number of rows to write
- **post_vertex_slice** (*Slice*) – The slice of the post vertex to generate for
- **n_synapse_types** (*int*) – The number of synapse types
- **max_n_synapses** (*int*) – The maximum number of synapses to generate

Returns (ff_data, ff_size)

Return type tuple(list(ndarray), ndarray)

get_value (*key*)

Get a property

Parameters **key** (*str*) – the name of the property

Return type Any or float or int or list(float) or list(int)

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

Return type str

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

Return type bool

mark_no_changes()
Marks the point after which changes are reported. Immediately after calling this method, requires_mapping should return False.

merge(*synapse_dynamics*)
Merge with the given synapse_dynamics and return the result, or error if merge is not possible

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

Return type *AbstractSynapseDynamics*

pad_to_length
The amount each row should pad to, or None if not specified

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*.

Parameters

- **post_vertex_slice** (*Slice*) –
- **n_synapse_types** (*int*) –
- **ff_size** (*ndarray*) –
- **ff_data** (*list(ndarray)*) –

requires_mapping
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_delay(*delay*)
Set the delay

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

Parameters

- **key** (*str*) – the name of the parameter to change
- **value** (*Any or float or int or list(float) or list(int)*) – the new value of the parameter to assign

weight
The weight of connections

write_parameters(*spec*, *region*, *machine_time_step*, *weight_scales*)
Write the synapse parameters to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **region** (*int*) – region ID

- **machine_time_step** (*int*) –
- **weight_scales** (*list(float)*) –

class spynnaker.pyNN.models.neuron.synapse_dynamics.**SynapseDynamicsSTDP**(*timing_dependence, weight_dependence, voltage_dependence=None, dendritic_delay_fraction=1.0, weight=<sphinx.ext.autodoc object>, delay=None, pad_to_length=None, backprop_delay=True*)

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

The dynamics of a synapse that changes over time using a Spike Timing Dependent Plasticity (STDP) rule.

Parameters

- **timing_dependence** (AbstractTimingDependence) –
- **weight_dependence** (AbstractWeightDependence) –
- **voltage_dependence** (*None*) – not supported
- **dendritic_delay_fraction** (*float*) – [0.5, 1.0]
- **weight** (*float*) –
- **delay** (*float or None*) – Use None to get the simulator default minimum delay.
- **pad_to_length** (*int or None*) –
- **backprop_delay** (*bool*) –

are_weights_signed()

Return type bool

backprop_delay

Settable.

Return type bool

changes_during_run

Determine if the synapses change during a run

Return type bool

delay

The delay of connections

dendritic_delay_fraction

Settable.

Return type float

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 ndarray(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** (*int*) – 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.

Parameters **fp_size** (*ndarray*) –

get_n_plastic_plastic_words_per_row(*pp_size*)
Get the number of plastic plastic words to be read from each row.

Parameters **pp_size** (*ndarray*) –

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*.

Parameters

- **pp_size** (*ndarray*) –
- **fp_size** (*ndarray*) –

get_n_words_for_plastic_connections(*n_connections*)

Parameters **n_connections** (*int*) –

Return type int

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*)

Parameters

- **n_neurons** (*int*) –
- **n_synapse_types** (*int*) –

Return type int

get_plastic_synaptic_data(*connections*, *connection_row_indices*, *n_rows*, *post_vertex_slice*, *n_synapse_types*, *max_n_synapses*)
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.

Parameters

- **connections** (*ndarray*) – The connections to get data for
- **connection_row_indices** (*ndarray*) – The row into which each connection should go
- **n_rows** (*int*) – The total number of rows
- **post_vertex_slice** (*Slice*) – The slice of the post vertex to get the connections for
- **n_synapse_types** (*int*) – The number of synapse types
- **max_n_synapses** (*int*) – The maximum number of synapses to generate

Returns (fp_data, pp_data, fp_size, pp_size)

Return type tuple(ndarray, ndarray, ndarray, ndarray)

get_provenance_data (*pre_population_label*, *post_population_label*)

Parameters

- **pre_population_label** (*str*) –
- **post_population_label** (*str*) –

Return type list(ProvenanceDataItem)

get_value (*key*)

Get a property

Parameters **key** (*str*) – the name of the property

Return type Any or float or int or list(float) or list(int)

get_vertex_executable_suffix ()

Return type str

get_weight_maximum (*connector*, *synapse_info*)

Get the maximum weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

get_weight_mean (*connector*, *synapse_info*)

Get the mean weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

get_weight_variance (*connector*, *weights*, *synapse_info*)

Get the variance in weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

Return type bool

mark_no_changes()
Marks the point after which changes are reported. Immediately after calling this method, requires_mapping should return False.

merge(*synapse_dynamics*)
Merge with the given synapse_dynamics and return the result, or error if merge is not possible

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

Return type *AbstractSynapseDynamics*

pad_to_length
The amount each row should pad to, or None if not specified

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*.

Parameters

- **post_vertex_slice** (*Slice*) –
- **n_synapse_types** (*int*) –
- **pp_size** (*ndarray*) – 1D
- **pp_data** (*ndarray*) – 2D
- **fp_size** (*ndarray*) – 1D
- **fp_data** (*ndarray*) – 2D

Returns array with columns source, target, weight, delay

Return type ndarray

requires_mapping
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_delay(*delay*)
Set the delay

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

Parameters

- **key** (*str*) – the name of the parameter to change
- **value** (*Any or float or int or list(float) or list(int)*) – the new value of the parameter to assign

timing_dependence

Return type *AbstractTimingDependence*

weight
The weight of connections

weight_dependence

Return type *AbstractTimingDependence*

write_parameters (*spec*, *region*, *machine_time_step*, *weight_scales*)

Parameters

- **spec** (`DataSpecificationGenerator`) –
- **region** (`int`) – region ID
- **machine_time_step** (`int`) –
- **weight_scales** (`list (float)`) –

class spynnaker.pyNN.models.neuron.synapse_dynamics.**SynapseDynamicsStructuralCommon**
Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.abstract_synapse_dynamics_structural.AbstractSynapseDynamicsStructural

PAIR_ERROR = 'Only one Projection between each pair of Populations can use structural p

check_initial_delay (*max_delay_ms*)
Check that delays can be done without delay extensions

Parameters **max_delay_ms** (`float`) – The maximum delay supported, in milliseconds

Raises **Exception** – if the delay is out of range

connections
initial connectivity as defined via connector

Return type dict

get_parameter_names ()

Return type list(str)

get_seeds (*app_vertex=None*)
Generate a seed for the RNG on chip that is the same for all of the cores that have perform structural updates.

It should be different between application vertices but the same for the same app_vertex. It should be different every time called with None.

Parameters **app_vertex** (`ApplicationVertex or None`) –

Returns list of random seed (4 words), generated randomly

Return type list(int)

get_structural_parameters_sdram_usage_in_bytes (*graph*, *vertex*, *n_neurons*)
Get the size of the structural parameters

Note: At the Application level this will be an estimate.

Parameters

- **graph** (`ApplicationGraph or MachineGraph`) – Graph at same level as vertex.
- **vertex** (`ApplicationVertex or MachineVertex`) – Vertex at the same level as the graph

- **n_neurons** (*int*) –

Returns the size of the parameters, in bytes

Return type int

Raises **PacmanInvalidParameterException** –

get_vertex_executable_suffix()

Return type str

is_same_as(*synapse_dynamics*)

Parameters **synapse_dynamics** (SynapseDynamicsStructuralCommon) –

Return type bool

p_rew

The period of rewiring.

Returns The period of rewiring

Return type float

write_structural_parameters(*spec, region, machine_time_step, weight_scales, machine_graph, machine_vertex, routing_info, synaptic_matrices*)

Write structural plasticity parameters

Parameters

- **spec** (*DataSpecificationGenerator*) – the data spec
- **region** (*int*) – region ID
- **machine_time_step** (*int*) – the duration of a machine time step (ms)
- **weight_scales** (*ndarray or list(float)*) – scaling the weights
- **machine_graph** (*MachineGraph*) – Full machine level network
- **machine_vertex** (AbstractPopulationVertex) – the vertex for which data specs are being prepared
- **routing_info** (*RoutingInfo*) – All of the routing information on the network
- **synaptic_matrices** (SynapticMatrices) – The synaptic matrices for this vertex

class spynnaker.pyNN.models.neuron.synapse_dynamics.**SynapseDynamicsStructuralStatic**(*partner_s formation, elimination, f_rew=10 initial_weig initial_delay s_max=3 with_repl seed=Nor weight=< object>, delay=None*

Bases: `spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_static.SynapseDynamicsStatic`, `spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_common.SynapseDynamicsStructuralCommon`

Class that enables synaptic rewiring in the absence of STDP.

It acts as a wrapper around SynapseDynamicsStatic, meaning that rewiring can operate in parallel with static synapses.

Written by Petrut Bogdan.

Parameters

- **partner_selection** (`AbstractPartnerSelection`) – The partner selection rule
- **formation** (`AbstractFormation`) – The formation rule
- **elimination** (`AbstractElimination`) – The elimination rule
- **f_rew** (*float*) – How many rewiring attempts will be done per second.
- **initial_weight** (*float*) – Weight assigned to a newly formed connection
- **initial_delay** (*float or (float, float)*) – Delay assigned to a newly formed connection; a single value means a fixed delay value, or a tuple of two values means the delay will be chosen at random from a uniform distribution between the given values
- **s_max** (*int*) – Maximum fan-in per target layer neuron
- **with_replacement** (*bool*) – If set to True (default), a new synapse can be formed in a location where a connection already exists; if False, then it must form where no connection already exists
- **seed** (*int*) – seed the random number generators
- **weight** (*float*) – The weight of connections formed by the connector
- **delay** (*float or None*) – The delay of connections formed by the connector Use `None` to get the simulator default minimum delay.

changes_during_run
Determine if the synapses change during a run

Return type bool

connections
initial connectivity as defined via connector

Return type dict

elimination
The elimination rule

Return type *AbstractElimination*

f_rew
The frequency of rewiring

Return type float

formation
The formation rule

Return type *AbstractFormation*

generate_on_machine()
Determines if this instance should be generated on the machine.

Default implementation returns True

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.

Parameters **n_connections** (*int*) –

Return type int

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_seeds(*app_vertex=None*)
Generate a seed for the RNG on chip that is the same for all of the cores that have perform structural updates.

It should be different between application vertices but the same for the same app_vertex. It should be different every time called with None.

Parameters **app_vertex** (*ApplicationVertex or None*) –

Returns list of random seed (4 words), generated randomly

Return type list(int)

get_vertex_executable_suffix()
Get the executable suffix for a vertex for this dynamics

Return type str

get_weight_maximum(*connector*, *synapse_info*)
Get the maximum weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

get_weight_mean (*connector*, *synapse_info*)
Get the mean weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

get_weight_variance (*connector*, *weights*, *synapse_info*)
Get the variance in weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

initial_delay
The delay of a formed connection

Return type float or (float, float)

initial_weight
The weight of a formed connection

Return type float

is_same_as (*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

Return type bool

merge (*synapse_dynamics*)
Merge with the given synapse_dynamics and return the result, or error if merge is not possible

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

Return type *AbstractSynapseDynamics*

partner_selection
The partner selection rule

Return type *AbstractPartnerSelection*

s_max
The maximum number of synapses

Return type int

seed
The seed to control the randomness

set_connections (*connections*, *post_vertex_slice*, *app_edge*, *synapse_info*, *machine_edge*)
Set connections for structural plasticity

Parameters

- **connections** (*ndarray*) –
- **post_vertex_slice** (*Slice*) –

- **app_edge** (ProjectionApplicationEdge) –
- **synapse_info** (SynapseInformation) –
- **machine_edge** (*MachineEdge*) –

set_projection_parameter (*param*, *value*)

Parameters

- **param** (*str*) –
- **value** –

with_replacement

Whether to allow replacement when creating synapses

Return type bool

class spynnaker.pyNN.models.neuron.synapse_dynamics.**SynapseDynamicsStructuralSTDP** (*partner_sele for- ma- tion, elim- i- na- tion, tim- ing_depende weight_depe volt- age_depende den- dritic_delay_ f_rew=1000(ini- tial_weight= ini- tial_delay=1 s_max=32, with_replace seed=None, weight=<spy ob- ject>, de- lay=None, back- prop_delay=*

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_stdp.SynapseDynamicsSTDP, spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_common.SynapseDynamicsStructuralCommon

Class that enables synaptic rewiring in the presence of STDP.

It acts as a wrapper around SynapseDynamicsSTDP, meaning rewiring can operate in parallel with STDP synapses.

Written by Petrut Bogdan.

Parameters

- **partner_selection** (AbstractPartnerSelection) – The partner selection rule
- **formation** (AbstractFormation) – The formation rule
- **elimination** (AbstractElimination) – The elimination rule
- **timing_dependence** (AbstractTimingDependence) – The STDP timing dependence rule
- **weight_dependence** (AbstractWeightDependence) – The STDP weight dependence rule
- **voltage_dependence** (*None*) – The STDP voltage dependence (unsupported)
- **dendritic_delay_fraction** (*float*) – The STDP dendritic delay fraction
- **f_rew** (*float*) – How many rewiring attempts will be done per second.
- **initial_weight** (*float*) – Weight assigned to a newly formed connection
- **initial_delay** (*float or tuple(float, float)*) – Delay assigned to a newly formed connection; a single value means a fixed delay value, or a tuple of two values means the delay will be chosen at random from a uniform distribution between the given values
- **s_max** (*int*) – Maximum fan-in per target layer neuron
- **with_replacement** (*bool*) – If set to True (default), a new synapse can be formed in a location where a connection already exists; if False, then it must form where no connection already exists
- **seed** (*int or None*) – seed for the random number generators
- **weight** (*float*) – The weight of connections formed by the connector
- **delay** (*float or None*) – The delay of connections formed by the connector Use `None` to get the simulator default minimum delay.

connections
initial connectivity as defined via connector

Return type dict

elimination
The elimination rule

Return type *AbstractElimination*

f_rew
The frequency of rewiring

Return type float

formation
The formation rule

Return type *AbstractFormation*

generate_on_machine()
Determines if this instance should be generated on the machine.

Default implementation returns True

Return type bool

get_n_words_for_plastic_connections(*n_connections*)

Parameters **n_connections** (*int*) –

Return type int

get_parameter_names()
Get the parameter names available from the synapse dynamics components

Return type iterable(str)

get_seeds(*app_vertex=None*)
Generate a seed for the RNG on chip that is the same for all of the cores that have perform structural updates.

It should be different between application vertices but the same for the same app_vertex. It should be different every time called with None.

Parameters **app_vertex** (*ApplicationVertex or None*) –

Returns list of random seed (4 words), generated randomly

Return type list(int)

get_vertex_executable_suffix()

Return type str

get_weight_maximum(*connector*, *synapse_info*)
Get the maximum weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

get_weight_mean(*connector*, *synapse_info*)
Get the mean weight for the synapses

Parameters

- **connector** (AbstractConnector) –
- **weights** (*ndarray*) –

initial_delay
The delay of a formed connection

Return type float or (float, float)

initial_weight
The weight of a formed connection

Return type float

is_same_as(*synapse_dynamics*)
Determines if this synapse dynamics is the same as another

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

Return type bool

merge(*synapse_dynamics*)
Merge with the given synapse_dynamics and return the result, or error if merge is not possible

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

Return type *AbstractSynapseDynamics*

partner_selection
The partner selection rule

Return type *AbstractPartnerSelection*

s_max
The maximum number of synapses

Return type int

seed
The seed to control the randomness

set_connections (*connections*, *post_vertex_slice*, *app_edge*, *synapse_info*, *machine_edge*)
Set connections for structural plasticity

Parameters

- **connections** (*ndarray*) –
- **post_vertex_slice** (*Slice*) –
- **app_edge** (ProjectionApplicationEdge) –
- **synapse_info** (SynapseInformation) –
- **machine_edge** (*MachineEdge*) –

set_projection_parameter (*param*, *value*)

Parameters

- **param** (*str*) –
- **value** –

with_replacement
Whether to allow replacement when creating synapses

Return type bool

spynnaker.pyNN.models.neuron.synapse_types package

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** (*list (DataType)*) – 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

Parameters

- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – $\tau^{syn}_{e_1}$
- **tau_syn_E2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $\tau^{syn}_{e_2}$
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_{i}
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – $I^{syn}_{e_1}$
- **isyn_exc2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $I^{syn}_{e_2}$
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_{i}

add_parameters(*parameters*)

Add the initial values of the parameters to the parameter holder

Parameters parameters (*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 (*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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*int*) – The time to be advanced in one call to the update of this component
- **ts** – machine time step

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

Return type list(int or float or list(int) or list(float) 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** (*list (list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – 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

Parameters

- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_e
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_i
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_e
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_i

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

Parameters **parameters** (*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** (*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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*int*) – The time to be advanced in one call to the update of this component
- **ts** – machine time step

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

Return type list(int or float or list(int) or list(float) 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** (*list(list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – 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

Parameters

- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_e
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_i

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

Parameters **parameters** (*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** (*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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*float*) – The time to be advanced in one call to the update of this component

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

Return type list(int or float or list(int) or list(float) 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** (*list (list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update

- **state_variables** (*RangeDictionary*) – 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

Parameters

- **exc_response** (*float, iterable(float), RandomDistribution or (mapping) function*) – $response_e^{\text{linear}}$
- **exc_exp_response** (*float, iterable(float), RandomDistribution or (mapping) function*) – $response_e^{\text{exponential}}$
- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_e^{syn}
- **inh_response** (*float, iterable(float), RandomDistribution or (mapping) function*) – $response_i^{\text{linear}}$
- **inh_exp_response** (*float, iterable(float), RandomDistribution or (mapping) function*) – $response_i^{\text{exponential}}$
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_i^{syn}

add_parameters(*parameters*)

Add the initial values of the parameters to the parameter holder

Parameters **parameters** (*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** (*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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*int*) – The time to be advanced in one call to the update of this component
- **ts** – machine time step

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

Return type list(int or float or list(int) or list(float) 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** (*list (list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

class spynnaker.pyNN.models.neuron.synapse_types.**SynapseTypeSEMD**(*tau_syn_E*, *tau_syn_E2*, *tau_syn_I*, *isyn_exc*, *isyn_exc2*, *isyn_inh*, *multiplicator*, *exc2_old*, *scaling_factor*)

Bases: `spynnaker.pyNN.models.neuron.synapse_types.abstract_synapse_type.AbstractSynapseType`

Parameters

- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – $\tau^{syn}_{e_1}$
- **tau_syn_E2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $\tau^{syn}_{e_2}$
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_{i}
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – $I^{syn}_{e_1}$
- **isyn_exc2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $I^{syn}_{e_2}$
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_{i}
- **multiplicator** (*float, iterable(float), RandomDistribution or (mapping) function*) –
- **exc2_old** (*float, iterable(float), RandomDistribution or (mapping) function*) –
- **scaling_factor** (*float, iterable(float), RandomDistribution or (mapping) function*) –

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

Parameters parameters (*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 (*RangeDictionary*) – A holder of the state variables

exc2_old

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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*int*) – The time to be advanced in one call to the update of this component
- **ts** – machine time step

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

Return type list(int or float or list(int) or list(float) 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

multiplicator

scaling_factor

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

spynnaker.pyNN.models.neuron.threshold_types package

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** (*list (DataType)*) – 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.

Parameters **v_thresh** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{thresh}

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

Parameters **parameters** (*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** (*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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*float*) – The time to be advanced in one call to the update of this component

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

Return type list(int or float or list(int) or list(float) 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** (*list (list)*) – The values read from the machine, one for each struct element
- **parameters** (*RangeDictionary*) – The holder of the parameters to update
- **state_variables** (*RangeDictionary*) – The holder of the state variables to update

v_thresh
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.

Habenschuss S, Jonke Z, Maass W. Stochastic computations in cortical microcircuit models. *PLoS Computational Biology*. 2013;9(11):e1003311. doi:10.1371/journal.pcbi.1003311

Parameters

- **du_th** (*float, iterable(float), RandomDistribution or (mapping) function*) – du_{thresh}
- **tau_th** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{thresh}
- **v_thresh** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{thresh}

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

Parameters **parameters** (*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** (*RangeDictionary*) – A holder of the state variables

du_th
du_{thresh}

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** (*RangeDictionary*) – The holder of the parameters
- **state_variables** (*RangeDictionary*) – The holder of the state variables
- **vertex_slice** (*Slice*) – The slice of variables being retrieved
- **ts** (*float*) – The time to be advanced in one call to the update of this component

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

Return type list(int or float or list(int) or list(float) 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
τ_{thresh}

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

v_thresh
V_{thresh}

Submodules

spynnaker.pyNN.models.neuron.key_space_tracker module

class spynnaker.pyNN.models.neuron.key_space_tracker.**KeySpaceTracker**
Bases: pacman.utilities.algorithm_utilities.element_allocator_algorithm.ElementAllocatorAlgorithm

Tracks keys used to determine key overlap

allocate_keys(*r_info*)

Allocate all the keys in the routing information NOTE assumes masks are all 1s followed by all 0s

Parameters **r_info** (*PartitionRoutingInfo*) – The routing information to add

static count_trailing_0s(*mask*)

Count bitwise zeros at the LSB end of a number NOTE assumes a 32-bit number

Parameters **mask** – The mask to be checked

is_allocated(*key*, *n_keys*)

Determine if any of the keys in the mask are allocated NOTE assumes mask is all 1s followed by all 0s

Parameters

- **key** (*int*) – The key at the start of the allocation
- **n_keys** (*int*) – The number of keys to check

Return type bool

spynnaker.pyNN.models.neuron.master_pop_table module

class spynnaker.pyNN.models.neuron.master_pop_table.**MasterPopTableAsBinarySearch**

Bases: object

Master population table, implemented as binary search master.

MAX_ROW_LENGTH_ERROR_MSG = 'Only rows of up to 256 entries are allowed'

OUT_OF_RANGE_ERROR_MESSAGE = 'Address {} is out of range for this population table!'

UPPER_BOUND_FUDGE = 2

add_application_entry(*block_start_addr*, *row_length*, *key_and_mask*, *core_mask*, *core_shift*, *n_neurons*)

Add an entry for an application-edge to the population table.

Parameters

- **block_start_addr** (*int*) – where the synaptic matrix block starts
- **row_length** (*int*) – how long in words each row is
- **key_and_mask** (*BaseKeyAndMask*) – the key and mask for this master pop entry
- **core_mask** (*int*) – Mask for the part of the key that identifies the core
- **core_shift** (*int*) – The shift of the mask to get to the core_mask
- **n_neurons** (*int*) – The number of neurons in each machine vertex (bar the last)
- **is_single** (*bool*) – 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

Raises ***SynapticConfigurationException*** – If a bad address is used.

add_invalid_entry(*key_and_mask*, *core_mask=0*, *core_shift=0*, *n_neurons=0*)

Add an entry to the table that doesn't point to anywhere. Used to keep indices in synchronisation between e.g. normal and delay entries and between entries on different cores.

Parameters

- **key_and_mask** (*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
- **core_mask** (*int*) – Mask for the part of the key that identifies the core
- **core_shift** (*int*) – The shift of the mask to get to the core_mask
- **n_neurons** (*int*) – The number of neurons in each machine vertex (bar the last)

Returns The index of the added entry

Return type int

add_machine_entry (*block_start_addr*, *row_length*, *key_and_mask*, *is_single=False*)
Add an entry for a machine-edge to the population table.

Parameters

- **block_start_addr** (*int*) – where the synaptic matrix block starts
- **row_length** (*int*) – how long in words each row is
- **key_and_mask** (*BaseKeyAndMask*) – the key and mask for this master pop entry
- **is_single** (*bool*) – 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

Raises *SynapticConfigurationException* – If a bad address is used.

finish_master_pop_table (*spec*, *master_pop_table_region*)
Complete the master pop table in the data specification.

Parameters

- **spec** (*DataSpecificationGenerator*) – the data specification to write the master pop entry to
- **master_pop_table_region** (*int*) – the region to which the master pop table is being stored

get_allowed_row_length (*row_length*)

Parameters **row_length** (*int*) – the row length being considered

Returns the row length available

Return type int

Raises *SynapseRowTooBigException* – If the row won't fit

get_master_population_table_size (*in_edges*)
Get the size of the master population table in SDRAM.

Parameters **in_edges** (*iterable(ApplicationEdge)*) – The edges arriving at the vertex that are to be handled by this table

Returns the size the master pop table will take in SDRAM (in bytes)

Return type int

get_next_allowed_address (*next_address*)
Get the next allowed address.

Parameters **next_address** (*int*) – The next address that would be used

Returns The next address that can be used following next_address

Return type int

Raises *SynapticConfigurationException* – if the address is out of range

initialise_table ()
Initialise the master pop data structure.

max_core_mask
The maximum core mask supported when n_neurons is > 0; this is the maximum number of cores that can be supported in a joined mask.

Return type int

max_index
The maximum index of a synaptic connection

Return type int

max_n_neurons_per_core
The maximum number of neurons per core supported when a core-mask is > 0.

Return type int

write_padding(*spec*, *next_block_start_address*)
Write padding to the data spec needed between blocks to align addresses correctly.

Parameters

- **spec** (*DataSpecificationGenerator*) – The spec to write to
- **next_block_start_address** (*int*) – The address we are starting at

Returns The address we finish at after the padding

Return type int

spynnaker.pyNN.models.neuron.synapse_io module

class spynnaker.pyNN.models.neuron.synapse_io.**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.

Parameters

- **undelayed_max_n_synapses** (*int*) – Maximum number of synapses in a row of the undelayed matrix
- **delayed_max_n_synapses** (*int*) – Maximum number of synapses in a row of the delayed matrix
- **undelayed_max_bytes** (*int*) – Maximum number of bytes, including headers, in a row of the undelayed matrix, or 0 if no synapses
- **delayed_max_bytes** (*int*) – Maximum number of bytes, including headers, in a row of the delayed matrix, or 0 if no synapses
- **undelayed_max_words** (*int*) – Maximum number of words, excluding headers, in a row of the undelayed matrix
- **delayed_max_words** (*int*) – Maximum number of words, excluding headers, in a row of the delayed matrix

delayed_max_bytes
Maximum number of bytes, including headers, in a row of the delayed matrix

Return type int

delayed_max_n_synapses
Maximum number of synapses in a row of the delayed matrix

Return type int

delayed_max_words
Maximum number of words, excluding headers, in a row of the undelayed matrix

Return type int

undelayed_max_bytes
Maximum number of bytes, including headers, in a row of the undelayed matrix

Return type int

undelayed_max_n_synapses
Maximum number of synapses in a row of the undelayed matrix

Return type int

undelayed_max_words
Maximum number of words, excluding headers, in a row of the undelayed matrix

Return type int

class spynnaker.pyNN.models.neuron.synapse_io.**SynapseIORowBased**
Bases: object

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.

convert_to_connections(*synapse_info*, *pre_vertex_slice*, *post_vertex_slice*, *max_row_length*, *n_synapse_types*, *weight_scales*, *data*, *machine_time_step*, *delayed*, *post_vertex_max_delay_ticks*)
Read the synapses for a given projection synapse information object out of the given data and convert to connection data

Parameters

- **synapse_info** (SynapseInformation) – The synapse information of the synapses
- **pre_vertex_slice** (*Slice*) – The slice of the source neurons of the synapses in the data
- **post_vertex_slice** (*Slice*) – The slice of the target neurons of the synapses in the data
- **max_row_length** (*int*) – The length of each row in the data
- **n_synapse_types** (*int*) – The number of synapse types in total
- **weight_scales** (*list(float)*) – The weight scaling of each synapse type
- **data** (*bytearray*) – The raw data containing the synapses
- **machine_time_step** (*int*) – The time step of the simulation
- **delayed** (*bool*) – True if the data should be considered delayed
- **post_vertex_max_delay_ticks** (*int*) – max delayed ticks supported from post vertex

Returns The connections read from the data; the dtype is AbstractSynapseDynamics.NUMPY_CONNECTORS_DTYPE

Return type ndarray

get_block_n_bytes (*max_row_n_words, n_rows*)
Get the number of bytes in a block

Parameters

- **max_row_n_words** (*int*) – The maximum row length in words, excluding headers
- **n_rows** (*int*) – The number of rows in the block

Return type int

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

Parameters

- **synapse_info** (SynapseInformation) – The synapse information to get the row data for
- **post_vertex_slice** (*Slice*) – The slice of the machine vertex being represented
- **n_delay_stages** (*int*) – The number of delay stages on the edge
- **population_table** (MasterPopTableAsBinarySearch) – The population table to be used
- **machine_time_step** (*int*) – The time step of the simulation
- **in_edge** (ProjectionApplicationEdge) – The incoming edge on which the synapse information is held

Return type *MaxRowInfo*

Raises ***SynapseRowTooBigException*** – If the synapse information can't be represented

get_maximum_delay_supported_in_ms (*machine_time_step, post_vertex_max_delay_ticks*)
Get the maximum delay supported by the synapse representation before extensions are required, or None if any delay is supported

Parameters

- **machine_time_step** (*int*) – The time step of the simulation
- **post_vertex_max_delay_ticks** (*int*) – post vertex max delay

Return type int

get_synapses (*synapse_info, n_delay_stages, n_synapse_types, weight_scales, machine_edge, max_row_info, gen_undelayed, gen_delayed, machine_time_step, app_edge*)
Get the synapses as an array of words for non-delayed synapses and an array of words for delayed synapses. This is used to prepare information for *deployment to SpiNNaker*.

Parameters

- **synapse_info** (SynapseInformation) – The synapse information to convert to synapses
- **n_delay_stages** (*int*) – The number of delay stages in total to be represented
- **n_synapse_types** (*int*) – The number of synapse types in total to be represented

- **weight_scales** (*list (float)*) – The scaling of the weights for each synapse type
- **machine_edge** (*MachineEdge*) – The incoming machine edge that the synapses are on
- **machine_time_step** (*int*) – The machine time step of the sim.
- **app_edge** (ProjectionApplicationEdge) –
- **max_row_info** (MaxRowInfo) – The maximum row information for the synapses
- **gen_undelayed** (*bool*) – Whether to generate undelayed data
- **gen_delayed** (*bool*) – Whether to generate delayed data

Returns

(row_data, delayed_row_data, delayed_source_ids, stages) where:

- **row_data is the undelayed connectivity data arranged into a** row per source, each row the same length
- **delayed_row_data is the delayed connectivity data arranged** into a row per source per delay stage, each row the same length
- **delayed_source_ids is the machine-vertex-local source neuron** id of each connection of the delayed vertices
- stages is the delay stage of each delayed connection

Return type tuple(ndarray, ndarray, ndarray, ndarray)

read_all_synapses (*data, delayed_data, synapse_info, n_synapse_types, weight_scales, machine_edge, max_row_info*)

Read the synapses for a given projection synapse information object out of the given delayed and undelayed data.

Parameters

- **data** (*bytearray*) – The raw data containing the undelayed synapses
- **delayed_data** (*bytearray*) – The raw data containing the delayed synapses
- **synapse_info** (SynapseInformation) – The synapse info that generated the synapses
- **n_synapse_types** (*int*) – The total number of synapse types available
- **weight_scales** (*list (float)*) – A weight scale for each synapse type
- **machine_edge** (*MachineEdge*) – The incoming machine edge that the synapses were generated from
- **max_row_info** (MaxRowInfo) – The maximum information for each of the rows

Returns The connections read from the data; the dtype is AbstractSynapseDynamics.NUMPY_CONNECTORS_DTYPE

Return type ndarray

spynnaker.pyNN.models.neuron.synaptic_matrices module

class spynnaker.pyNN.models.neuron.synaptic_matrices.**SynapticMatrices**(*post_vertex_slice*, *n_synapse_types*, *all_single_syn_sz*, *synapse_io*, *synaptic_matrix_region*, *direct_matrix_region*, *poptable_region*)

Bases: object

Handler of synaptic matrices for a core of a population vertex

Parameters

- **post_vertex_slice** (*Slice*) – The slice of the post vertex that these matrices are for
- **n_synapse_types** (*int*) – The number of synapse types available
- **all_single_syn_sz** (*int*) – The space available for "direct" or "single" synapses
- **synapse_io** (SynapseIORowBased) – How to read and write synapses
- **synaptic_matrix_region** (*int*) – The region where synaptic matrices are stored
- **direct_matrix_region** (*int*) – The region where "direct" or "single" synapses are stored
- **poptable_region** (*int*) – The region where the population table is stored

clear_connection_cache()

Clear any values read from the machine

gen_on_machine

Whether any matrices need to be generated on the machine

Return type bool

get_connections_from_machine(*transceiver*, *placement*, *app_edge*, *synapse_info*)

Get the synaptic connections from the machine

Parameters

- **transceiver** (*Transceiver*) – Used to read the data from the machine
- **placement** (*Placement*) – Where the vertices are on the machine
- **app_edge** (ProjectionApplicationEdge) – The application edge of the projection
- **synapse_info** (SynapseInformation) – The synapse information of the projection

Returns A list of arrays of connections, each with dtype AbstractSynapseDynamics.NUMPY_CONNECTORS_DTYPE

Return type ndarray

get_index(*app_edge*, *synapse_info*, *machine_edge*)

Get the index of an incoming projection in the population table

Parameters

- **app_edge** (ProjectionApplicationEdge) – The application edge of the projection
- **synapse_info** (SynapseInformation) – The synapse information of the projection
- **machine_edge** (*MachineEdge*) – The machine edge to get the index of

host_generated_block_addr
The address within the synaptic region after the last block written by the on-host synaptic generation

on_chip_generated_block_addr
The address within the synaptic region after the last block reserved for the on-machine synaptic generation

read_generated_connection_holders (*transceiver*, *placement*)

Fill in any pre-run connection holders for data which is generated on the machine, after it has been generated

Parameters

- **transceiver** (*Transceiver*) – How to read the data from the machine
- **placement** (*Placement*) – where the data is to be read from

size (*app_edges*)
The size required by all parts of the matrices

Parameters **app_edges** (*iterable(ApplicationEdge)*) – The incoming application edges

Return type int

synapses_size (*app_edges*)
The size of the synaptic blocks in bytes

Parameters **app_edges** (*iterable(ApplicationEdge)*) – The incoming application edges

Return type int

write_synaptic_matrix_and_master_population_table (*spec*, *machine_vertex*, *all_syn_block_sz*, *weight_scales*, *routing_info*, *machine_graph*, *machine_time_step*)

Simultaneously generates both the master population table and the synaptic matrix.

Parameters

- **spec** (*DataSpecificationGenerator*) – The spec to write to
- **machine_vertex** (*MachineVertex*) – The machine vertex to write for
- **all_syn_block_sz** (*int*) – The size in bytes of the space reserved for synapses
- **weight_scales** (*list(float)*) – The weight scale of each synapse
- **routing_info** (*RoutingInfo*) – The routing information for all edges
- **machine_graph** (*MachineGraph*) – The machine graph
- **machine_time_step** (*float*) – sim machine time step

Returns A list of generator data to be written elsewhere

Return type list(GeneratorData)

spynnaker.pyNN.models.neuron.synaptic_matrix module

class spynnaker.pyNN.models.neuron.synaptic_matrix.**SynapticMatrix**(*synapse_io*, *poptable*, *synapse_info*, *machine_edge*, *app_edge*, *n_synapse_types*, *max_row_info*, *routing_info*, *delay_routing_info*, *weight_scales*, *all_syn_block_sz*, *all_single_syn_sz*)

Bases: `object`

Synaptic matrix/matrices for an incoming machine edge

Parameters

- **synapse_io** (`SynapseIORowBased`) – The reader and writer of synapses
- **poptable** (`MasterPopTableAsBinarySearch`) – The master population table
- **synapse_info** (`SynapseInformation`) – The projection synapse information
- **machine_edge** (*MachineEdge*) – The projection machine edge
- **app_edge** (`ProjectionApplicationEdge`) – The projection application edge
- **n_synapse_types** (*int*) – The number of synapse types accepted
- **max_row_info** (`MaxRowInfo`) – Maximum row length information
- **routing_info** (*PartitionRoutingInfo*) – Routing information for the edge
- **delay_routing_info** (*PartitionRoutingInfo*) – Routing information for the delay edge if any
- **weight_scales** (*list(float)*) – Weight scale for each synapse type
- **all_syn_block_sz** – The space available for all synaptic matrices
- **all_single_syn_sz** (*int*) – The space available for "direct" or "single" synapses

clear_connection_cache()

Clear the saved connections

delay_index

The index of the delayed matrix within the master population table

Return type int

get_generator_data(*syn_mat_offset*, *d_mat_offset*, *max_delay_per_stage*, *machine_time_step*)

Get the generator data for this matrix

Parameters

- **syn_mat_offset** (*int*) – The synaptic matrix offset to write the data to
- **machine_time_step** (*float*) – the sim's machine time step.
- **d_mat_offset** (*int*) – The synaptic matrix offset to write the delayed data to
- **max_delay_per_stage** (*int*) – around of timer ticks each delay stage holds.

Return type GeneratorData

get_row_data (*machine_time_step*)
Generate the row data for a synaptic matrix from the description

Parameters **machine_time_step** (*float*) – the sim machine time step.

Returns The data and the delayed data

Return type tuple(ndarray or None, ndarray or None)

index
The index of the matrix within the master population table

Return type int

is_delayed
Is there a delay matrix?

Return type bool

is_direct (*single_addr*)
Determine if the given connection can be done with a "direct" synaptic matrix - this must have an exactly 1 entry per row

Parameters **single_addr** (*int*) – The current offset of the direct matrix

Returns A tuple of a boolean indicating if the matrix is direct and the next offset of the single matrix

Return type (bool, int)

next_app_delay_on_chip_address (*app_block_addr*, *max_app_addr*)
Allocate a machine-level address of a delayed matrix from within an app-level allocation

Parameters

- **app_block_addr** (*int*) – The current position in the application block
- **max_app_addr** (*int*) – The position of the end of the allocation

Returns The address after the allocation and the allocated address

Return type int, int

next_app_on_chip_address (*app_block_addr*, *max_app_addr*)
Allocate a machine-level address of a matrix from within an app-level allocation

Parameters

- **app_block_addr** (*int*) – The current position in the application block
- **max_app_addr** (*int*) – The position of the end of the allocation

Returns The address after the allocation and the allocated address

Return type int, int

next_delay_on_chip_address (*block_addr*)
Allocate an address for a delayed machine matrix and add it to the population table

Parameters **block_addr** (*int*) – The address at which to start the allocation

Returns The address after the allocation and the allocated address

Return type int, int

next_on_chip_address (*block_addr*)
Allocate an address for a machine matrix and add it to the population table

Parameters **block_addr** (*int*) – The address at which to start the allocation

Returns The address after the allocation and the allocated address

Return type int, int

read_connections (*transceiver*, *placement*, *synapses_address*, *single_address*)
Read the connections from the machine

Parameters

- **transceiver** (*Transceiver*) – How to read the data from the machine
- **placement** (*Placement*) – Where the matrix is on the machine
- **synapses_address** (*int*) – The base address of the synaptic matrix region
- **single_address** (*int*) – The base address of the "direct" or "single" matrix region

Returns A list of arrays of connections, each with dtype AbstractSynapseDynamics.NUMPY_CONNECTORS_DTYPE

Return type ndarray

write_delayed_machine_matrix (*spec*, *block_addr*, *row_data*)
Write a delayed matrix for an incoming machine vertex

Parameters

- **spec** (*DataSpecificationGenerator*) – The specification to write to
- **block_addr** (*int*) – The address in the synaptic matrix region to start writing at
- **row_data** (*ndarray*) – The data to write

Returns The updated block address

Return type int

write_machine_matrix (*spec*, *block_addr*, *single_synapses*, *single_addr*, *row_data*)
Write a matrix for the incoming machine vertex

Parameters

- **spec** (*DataSpecificationGenerator*) – The specification to write to
- **block_addr** (*int*) – The address in the synaptic matrix region to start writing at
- **single_addr** (*int*) – The address in the "direct" or "single" matrix to start at
- **single_synapses** (*list*) – A list of "direct" or "single" synapses to write to
- **row_data** (*ndarray*) – The data to write

Returns The updated block and single addresses

Return type tuple(int, int)

spynnaker.pyNN.models.neuron.synaptic_matrix_app module

class spynnaker.pyNN.models.neuron.synaptic_matrix_app.**SynapticMatrixApp**(*synapse_io*, *poptable*, *synapse_info*, *app_edge*, *n_synapse_types*, *all_single_syn_sz*, *post_vertex_slice*, *synaptic_matrix_region*, *direct_matrix_region*)

Bases: `object`

The synaptic matrix (and delay matrix if applicable) for an incoming app edge

Parameters

- **synapse_io** (`SynapseIORowBased`) – The reader and writer of synapses
- **poptable** (`MasterPopTableAsBinarySearch`) – The master population table
- **synapse_info** (`SynapseInformation`) – The projection synapse information
- **app_edge** (`ProjectionApplicationEdge`) – The projection application edge
- **n_synapse_types** (*int*) – The number of synapse types accepted
- **all_single_syn_sz** (*int*) – The space available for "direct" or "single" synapses
- **post_vertex_slice** (*Slice*) – The slice of the post-vertex the matrix is for
- **synaptic_matrix_region** (*int*) – The region where synaptic matrices are stored
- **direct_matrix_region** (*int*) – The region where "direct" or "single" synapses are stored

add_delayed_matrix_size(*addr*)

Add the bytes required by the delayed synaptic matrices

Parameters **addr** (*int*) – The initial address

Returns The final address after adding synapses

Return type int

add_matrix_size(*addr*)

Add the bytes required by the synaptic matrices

Parameters **addr** (*int*) – The initial address

Returns The final address after adding synapses

Return type int

can_generate_on_machine(*single_addr*)

Determine if an app edge can be generated on the machine

Parameters **single_addr** (*int*) – The address for "direct" or "single" synapses so far

Return type bool

clear_connection_cache()
Clear saved connections

generator_info_size
The number of bytes required by the generator information

Return type int

get_connections(*transceiver*, *placement*)
Get the connections for this matrix from the machine

Parameters

- **transceiver** (*Transceiver*) – How to read the data from the machine
- **placement** (*Placement*) – Where the matrix is on the machine

Returns A list of arrays of connections, each with dtype AbstractSynapseDynamics.NUMPY_CONNECTORS_DTYPE

Return type ndarray

get_delay_index(*machine_edge*)

Get the index in the master population table of the delayed matrix for a machine edge

Parameters machine_edge (*MachineEdge*) – The edge to get the index for

Return type int

get_index(*machine_edge*)

Get the index in the master population table of the matrix for a machine edge

Parameters machine_edge (*MachineEdge*) – The edge to get the index for

Return type int

read_generated_connection_holders(*transceiver*, *placement*)
Read any pre-run connection holders after data has been generated

Parameters

- **transceiver** (*Transceiver*) – How to read the data from the machine
- **placement** (*Placement*) – Where the matrix is on the machine

set_info(*all_syn_block_sz*, *app_key_info*, *delay_app_key_info*, *routing_info*, *weight_scales*, *m_edges*)

Set extra information that isn't necessarily available when the class is created.

Parameters

- **all_syn_block_sz** (*int*) – The space available for all synaptic matrices on the core
- **app_key_info** (*_AppKeyInfo*) – Application-level routing key information for undelayed vertices
- **delay_app_key_info** (*_AppKeyInfo*) – Application-level routing key information for delayed vertices
- **routing_info** (*RoutingInfo*) – Routing key information for all incoming edges
- **weight_scales** (*list(float)*) – Weight scale for each synapse edge

- **m_edges** (*list (MachineEdge)*) – The machine edges incoming to this vertex

write_matrix (*spec, block_addr, single_addr, single_synapses, machine_time_step*)
Write a synaptic matrix from host

Parameters

- **spec** (*DataSpecificationGenerator*) – The specification to write to
- **block_addr** (*int*) – The address in the synaptic matrix region to start writing at
- **single_addr** (*int*) – The address in the "direct" or "single" matrix to start at
- **single_synapses** (*list (int)*) – A list of "direct" or "single" synapses to write to
- **machine_time_step** (*float*) – the simulation machine time step

Returns The updated block_addr and single_addr

Return type tuple(int, int)

write_on_chip_matrix_data (*generator_data, block_addr, machine_time_step*)
Prepare to write a matrix using an on-chip generator

Parameters

- **generator_data** (*list (GeneratorData)*) – List of data to add to
- **block_addr** (*int*) – The address in the synaptic matrix region to start writing at
- **machine_time_step** (*float*) – the sim machine time step

Returns The updated block address

Return type int

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, drop_late_spikes, splitter*)

Bases: spinn_front_end_common.abstract_models.impl.tdma_aware_application_vertex.TDMAAwareApplicationVertex, 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 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, 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. Not actually abstract.

Parameters

- **n_neurons** (*int*) – The number of neurons in the population
- **label** (*str*) – The label on the population
- **constraints** (*list(AbstractConstraint)*) – Constraints on where a population's vertices may be placed.
- **max_atoms_per_core** (*int*) – The maximum number of atoms (neurons) per SpiNNaker core.
- **spikes_per_second** (*float or None*) – Expected spike rate
- **ring_buffer_sigma** (*float or None*) – How many SD above the mean to go for upper bound of ring buffer size; a good starting choice is 5.0. Given length of simulation we can set this for approximate number of saturation events.
- **incoming_spike_buffer_size** (*int or None*) –
- **drop_late_spikes** (*bool*) – control flag for dropping late packets.
- **neuron_impl** (AbstractNeuronImpl) – The (Python side of the) implementation of the neurons themselves.
- **pynn_model** (AbstractPyNNNeuronModel) – The PyNN neuron model that this vertex is working on behalf of.
- **splitter** (*None or AbstractSplitterCommon*) – splitter object

BYTES_TILL_START_OF_GLOBAL_PARAMETERS = 20

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

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

Parameters

- **variable** (*str*) – PyNN name of the variable
- **buffer_manager** (*BufferManager*) – the buffer manager object
- **placements** (*Placements*) – the placements object

Return type None

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

Parameters

- **buffer_manager** (*BufferManager*) – the buffer manager object
- **placements** (*Placements*) – the placements object

Return type None

conductance_based

Return type bool

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.

Return type dict(str, ..)

get_connections_from_machine(*transceiver*, *placements*, *app_edge*, *synapse_info*)

Get the connections from the machine post-run.

Parameters

- **transceiver** (*Transceiver*) – How to read the connection data
- **placements** (*Placements*) – Where the connection data is on the machine
- **app_edge** (`ProjectionApplicationEdge`) – The edge for which the data is being read
- **synapse_info** (`SynapseInformation`) – The specific projection within the edge

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

Get the recorded data

Parameters

- **variable** (*str*) – PyNN name of the variable
- **n_machine_time_steps** (*int*) –
- **placements** (*Placements*) –
- **buffer_manager** (*BufferManager*) –
- **machine_time_step** (*int*) – microseconds

Returns (data, recording_indices, sampling_interval)

Return type tuple(ndarray,list(int),float)

get_expected_n_rows(*n_machine_time_steps*, *sampling_rate*, *vertex*, *variable*)

Returns the number of expected rows for a given runtime

Parameters

- **n_machine_time_steps** (*int*) – map of vertex to steps.
- **sampling_rate** (*int*) – the sampling rate for this vertex
- **vertex** (*MachineVertex*) – the machine vertex
- **variable** (*str*) – the variable being recorded

Returns int the number of rows expected.

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 or without _init
- **selector** (*None or slice or int or list(bool) or list(int)*) – a description of the subrange to accept, or None for all. See: selector_to_ids()

Returns A list or an Object which act like a list

Return type iterable

get_neuron_sampling_interval(*variable*)

Returns the current sampling interval for this variable

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

Returns Sampling interval in microseconds

Return type float

get_outgoing_partition_constraints(*partition*)

Get constraints to be added to the given edge partition 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 PyNN names of variables this model is expected to collect

Return type list(str)

get_sdram_usage_for_neuron_params(*vertex_slice*)

Calculate the SDRAM usage for just the neuron parameters region.

Parameters **vertex_slice** (*Slice*) – the slice of atoms.

Returns The SDRAM required for the neuron region

get_spikes(*placements*, *buffer_manager*, *machine_time_step*)

Get the recorded spikes from the object

Parameters

- **placements** (*Placements*) – the placements object
- **buffer_manager** (*BufferManager*) – the buffer manager object
- **machine_time_step** (*int*) – the time step of the simulation, in microseconds

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

Return type ndarray(tuple(int,int))

get_spikes_sampling_interval()

Return the current sampling interval for spikes

Returns Sampling interval in microseconds

Return type float

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** (*str*) – the variable to find units from

Returns the units as a string.

Return type str

get_value (*key*)
Get a property

Parameters **key** (*str*) – the name of the property

Return type Any or float or int or list(float) or list(int) Get a property of the overall model.

incoming_spike_buffer_size

initialize (*variable*, *value*, *selector=None*)
Set the initial value of one of the state variables of the neurons in this population.

Parameters

- **variable** (*str*) – The name of the variable to set
- **value** (*float or int or Any*) – The value of the variable to set

initialize_parameters
The names of parameters that have default initial values.

Return type iterable(str)

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

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

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.

n_atoms
The number of atoms in the vertex

Return type int

n_profile_samples

neuron_impl

neuron_recorder

parameters

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

reset_ring_buffer_shifts()

reset_to_first_timestep()
Reset the object to first time step.

ring_buffer_sigma

set_has_run()
Set the flag has run so initialize only affects state variables

Return type None

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

Parameters

- **variable** (*str*) – PyNN name of the variable
- **new_state** (*bool*) –
- **sampling_interval** (*int or None*) –
- **indexes** (*list or None*) – Which indices are to be recorded (or None for all)

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** (*int or None*) – The interval at which spikes are recorded. Must be a whole multiple of the timestep. None will be taken as the timestep.
- **indexes** (*list(int) or None*) – The indexes of the neurons that will record spikes. If None the assumption is all neurons are recording

set_synapse_dynamics(*synapse_dynamics*)

Parameters **synapse_dynamics** (AbstractSynapseDynamics) –

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

Parameters

- **key** (*str*) – the name of the parameter to change

- **value** (*Any or float or int or list(float) or list(int) Set a property of the overall model.*) – the new value of the parameter to assign

spikes_per_second

state_variables

synapse_dynamics

Return type AbstractSynapseDynamics

synapse_manager

weight_scale

Return type float

class spynnaker.pyNN.models.neuron.**AbstractPyNNNeuronModel**(*model*)

Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

Parameters **model** (AbstractNeuronImpl) – The model implementation

create_vertex(*n_neurons, label, constraints, spikes_per_second, ring_buffer_sigma, incoming_spike_buffer_size, drop_late_spikes, splitter*)

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(AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type ApplicationVertex

default_population_parameters = {'drop_late_spikes': None, 'incoming_spike_buffer_siz

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=256*)

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

A neuron model that follows the sPyNNaker standard composed model pattern for point neurons.

Parameters

- **model_name** (*str*) – Name of the model.
- **binary** (*str*) – Name of the implementation executable.
- **neuron_model** (AbstractNeuronModel) – The model of the neuron soma
- **input_type** (AbstractInputType) – The model of synaptic input types
- **synapse_type** (AbstractSynapseType) – The model of the synapses' dynamics
- **threshold_type** (AbstractThresholdType) – The model of the firing threshold
- **additional_input_type** (AbstractAdditionalInput *or None*) – The model (if any) of additional environmental inputs

create_vertex(*n_neurons*, *label*, *constraints*, *spikes_per_second*, *ring_buffer_sigma*, *incoming_spike_buffer_size*, *n_steps_per_timestep*, *drop_late_spikes*, *splitter*)

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(AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type ApplicationVertex

default_population_parameters = {'drop_late_spikes': None, 'incoming_spike_buffer_size

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** (*list(int) or tuple(int) or None*) – A list of data fields to be returned
- **as_list** (*bool*) – 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** (*int*) – The number of atoms in the pre-vertex
- **n_post_atoms** (*int*) – The number of atoms in the post-vertex
- **connections** (*list(ndarray) or None*) – Any initial connections, as a numpy structured array of source, target, weight and delay
- **fixed_values** (*list(tuple(str,int)) or None*) – 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** (*callable(ConnectionHolder, None) or None*) – 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 (*ndarray*) – The connection to add, as a numpy structured array of source, target, weight and delay

connections
The connections stored

Return type list(ndarray)

finish ()
Finish adding connections

class spynnaker.pyNN.models.neuron.**PopulationMachineVertex** (*resources_required, recorded_region_ids, label, constraints, app_vertex, vertex_slice, drop_late_spikes, binary_file_name*)

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.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, spinn_front_end_common.interface.provenance.provides_provenance_data_from_machine_impl.ProvidesProvenanceDataFromMachineImpl, spinn_front_end_common.interface.profiling.abstract_has_profile_data.AbstractHasProfileData, spinn_front_end_common.abstract_models.abstract_supports_bit_field_generation.AbstractSupportsBitFieldGeneration, spinn_front_end_common.abstract_models.abstract_supports_bit_field_routing_compression.AbstractSupportsBitFieldRoutingCompression, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification, spynnaker.pyNN.models.abstract_models.abstract_synapse_expandable.AbstractSynapseExpandable, spinn_front_end_common.abstract_models.abstract_rewrites_data_specification.AbstractRewritesDataSpecification, spynnaker.pyNN.models.abstract_models.abstract_read_parameters_before_set.AbstractReadParametersBeforeSet

Parameters

- **resources_required** (*ResourceContainer*) –
- **recorded_region_ids** (*iterable(int)*) –
- **label** (*str*) –
- **drop_late_spikes** (*bool*) – control flag for dropping packets.
- **constraints** (*list(AbstractConstraint)*) –
- **app_vertex** (AbstractPopulationVertex) – The associated application vertex
- **vertex_slice** (*Slice*) – The slice of the population that this implements
- **binary_file_name** (*str*) – binary name to be run for this verte

BIT_FIELDS_NOT_READ = 'N bit fields not able to be read into DTCM'

BIT_FIELD_FILTERED_PACKETS = 'How many packets were filtered by the bitfield filterer.

DMA_COMPLETE = "DMA's that were completed"

class EXTRA_PROVENANCE_DATA_ENTRIES

Bases: `enum.Enum`

Entries for the provenance data generated by standard neuron models.

BIT_FIELD_FILTERED_COUNT = 10

BUFFER_OVERFLOW_COUNT = 2

The number of times there was a buffer overflow

CURRENT_TIMER_TIC = 3

The current timer tick

DMA_COMPLETES = 7

FAILED_TO_READ_BIT_FIELDS = 6

GHOST_POP_TABLE_SEARCHES = 5

INPUT_BUFFER_FILLED_SIZE = 13

INVALID_MASTER_POP_HITS = 9

MAX_BACKGROUND_QUEUED = 15

N_BACKGROUND_OVERLOADS = 16

N_LATE_SPIKES = 12

N_REWIRES = 11

PLASTIC_SYNAPTIC_WEIGHT_SATURATION_COUNT = 4

The number of times the plastic synapses saturated during weight calculation

PRE_SYNAPTIC_EVENT_COUNT = 0

The number of pre-synaptic events

SATURATION_COUNT = 1

The number of times the synapse arithmetic saturated

SPIKE_PROGRESSING_COUNT = 8

TDMA_MISSES = 14

GHOST_SEARCHES = 'Number of failed pop table searches'

INPUT_BUFFER_FULL_NAME = 'Times_the_input_buffer_lost_packets'

INVALID_MASTER_POP_HITS = 'Invalid Master Pop hits'

LAST_TIMER_TICK = 'Last_timer_tic_the_core_ran_to'

LAST_TIMER_TICK_NAME = 'Last_timer_tic_the_core_ran_to'

LOST_INPUT_BUFFER_PACKETS = 'Times_the_input_buffer_lost_packets'

N_ADDITIONAL_PROVENANCE_DATA_ITEMS = 17

N_RE_WIRES_NAME = 'Number_of_rewires'

PLASTIC_WEIGHT_SATURATION = 'Times_plastic_synaptic_weights_have_saturated'

SATURATED_PLASTIC_WEIGHTS_NAME = 'Times_plastic_synaptic_weights_have_saturated'

SATURATION_COUNT_NAME = 'Times_synaptic_weights_have_saturated'

SPIKES_PROCESSED = 'how many spikes were processed'

TOTAL_PRE_SYNAPTIC_EVENTS = 'Total_pre_synaptic_events'

TOTAL_PRE_SYNAPTIC_EVENT_NAME = 'Total_pre_synaptic_events'

bit_field_base_address(*transceiver*, *placement*)
Returns the SDRAM address for the bit field table data.

Parameters

- **transceiver** (*Transceiver*) –
- **placement** (*Placement*) –

Returns the SDRAM address for the bitfield address

Return type int

bit_field_builder_region(*transceiver*, *placement*)
returns the SDRAM address for the bit field builder data

Parameters

- **transceiver** (*Transceiver*) –
- **placement** (*Placement*) –

Returns the SDRAM address for the bitfield builder data

Return type int

gen_on_machine()
True if the synapses of a the slice of this vertex should be generated on the machine.

Note: The typical implementation for this method will be to ask the app_vertex's synapse_manager

Return type bool

generate_data_specification(*spec*, *placement*, *machine_time_step*, *time_scale_factor*, *application_graph*, *machine_graph*, *routing_info*, *data_n_time_steps*, *n_key_map*)
Generate a data specification.

Parameters

- **spec** (*DataSpecificationGenerator*) – The data specification to write to
- **placement** (*Placement*) – The placement the vertex is located at
- **machine_time_step** – (injected)
- **time_scale_factor** – (injected)
- **application_graph** – (injected)
- **machine_graph** – (injected)
- **routing_info** – (injected)
- **data_n_time_steps** – (injected)
- **n_key_map** – (injected)

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_profile_data(*transceiver*, *placement*)
Get the profile data recorded during simulation

Parameters

- **transceiver** (*Transceiver*) –
- **placement** (*Placement*) –

Return type 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

key_to_atom_map_region_base_address(*transceiver*, *placement*)
Returns the SDRAM address for the region that contains key-to-atom data.

Parameters

- **transceiver** (*Transceiver*) –
- **placement** (*Placement*) –

Returns the SDRAM address for the key-to-atom data

Return type int

static neuron_region_sdram_address(*placement*, *transceiver*)

read_generated_connection_holders(*transceiver*, *placement*)
Fill in the connection holders

Note: The typical implementation for this method will be to ask the app_vertex's synapse_manager

Parameters

- **transceiver** (*Transceiver*) – How the data is to be read
- **placement** (*Placement*) – Where the data is on the machine

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

Parameters

- **transceiver** (*Transceiver*) – the SpinnMan interface
- **placement** (*Placement*) – the placement of a vertex
- **vertex_slice** (*Slice*) – the slice of atoms for this vertex

Return type None

regeneratable_sdram_blocks_and_sizes(*transceiver*, *placement*)
Returns the SDRAM addresses and sizes for the cores' SDRAM that are available (borrowed) for generating bitfield tables.

Parameters

- **transceiver** (*Transceiver*) –
- **placement** (*Placement*) –

Returns list of tuples containing (the SDRAM address for the cores SDRAM address's for the core's SDRAM that can be used to generate bitfield tables loaded, and the size of memory chunks located there)

Return type list(tuple(int,int))

regenerate_data_specification(*spec*, *placement*, *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?

reload_required()
Return true if any data region needs to be reloaded

Return type bool

resources_required
The resources required by the vertex

Return type ResourceContainer

set_on_chip_generatable_area(*offset*, *size*)

set_reload_required(*new_value*)
Indicate that the regions have been reloaded

Parameters **new_value** – the new value

Return type None

class spynnaker.pyNN.models.neuron.**SynapticManager**(*n_synapse_types*, *ring_buffer_sigma*, *spikes_per_second*, *config*, *drop_late_spikes*)

Bases: object

Deals with synapses

Parameters

- **n_synapse_types** (*int*) – number of synapse types on a neuron (e.g., 2 for excitatory and inhibitory)
- **ring_buffer_sigma** (*float or None*) – How many SD above the mean to go for upper bound; a good starting choice is 5.0. Given length of simulation we can set this for approximate number of saturation events.
- **spikes_per_second** (*float or None*) – Estimated spikes per second
- **config** (*RawConfigParser*) – The system configuration
- **drop_late_spikes** (*bool*) – control flag for dropping late packets.

FUDGE = 0

INDEXS_DONT_MATCH_ERROR_MESSAGE = 'Delay index {} and normal index {} do not match'

NOT_EXACT_SLICES_ERROR_MESSAGE = 'The splitter {} is returning estimated slices during

NO_DELAY_EDGE_FOR_SRC_IDS_MESSAGE = 'Found delayed source IDs but no delay machine edg

STATIC_SYNAPSE_MATRIX_SDRAM_IN_BYTES = 8

TOO_MUCH_WRITTEN_SYNAPTIC_DATA = 'Too much synaptic memory has been written: {} of {}

changes_during_run

Whether the synapses being managed change during running.

Return type bool

clear_all_caches()

Clears all cached data in the case that a reset requires remapping which might change things

clear_connection_cache()

Flush the cache of connection information; needed for a second run

drop_late_spikes

gen_on_machine(*post_vertex_slice*)

True if the synapses should be generated on the machine

Parameters **post_vertex_slice** (*Slice*) – The slice of the vertex to determine the generation status of

Return type bool

get_connections_from_machine(*transceiver*, *placements*, *app_edge*, *synapse_info*)

Read the connections from the machine for a given projection

Parameters

- **transceiver** (*Transceiver*) – Used to read the data from the machine
- **placements** (*Placements*) – Where the vertices are on the machine

- **app_edge** (ProjectionApplicationEdge) – The application edge of the projection
- **synapse_info** (SynapseInformation) – The synapse information of the projection

Returns The connections from the machine, with dtype AbstractSynapseDynamics.NUMPY_CONNECTORS_DTYPE

Return type ndarray

get_dtcm_usage_in_bytes()

Return type int

get_n_cpu_cycles()

Return type int

get_sdram_usage_in_bytes(*post_vertex_slice*, *application_graph*, *app_vertex*)

Get the SDRAM usage of a slice of atoms of this vertex

Parameters

- **post_vertex_slice** (*Slice*) – The slice of atoms to get the size of
- **application_graph** (*ApplicationGraph*) – The application graph
- **app_vertex** (AbstractPopulationVertex) – The application vertex

Return type int

host_written_matrix_size(*post_vertex_slice*)

The size of the matrix written by the host for a given machine vertex

Parameters **post_vertex_slice** – The slice of the vertex to get the size of

Return type int

on_chip_written_matrix_size(*post_vertex_slice*)

The size of the matrix that will be written on the machine for a given machine vertex

Parameters **post_vertex_slice** – The slice of the vertex to get the size of

Return type int

read_generated_connection_holders(*transceiver*, *placement*)

Fill in any pre-run connection holders for data which is generated on the machine, after it has been generated

Parameters

- **transceiver** (*Transceiver*) – How to read the data from the machine
- **placement** (*Placement*) – where the data is to be read from

reset_ring_buffer_shifts()

Reset the ring buffer shifts; needed if projection data changes between runs

ring_buffer_sigma

The sigma in the estimation of the maximum summed ring buffer weights. Settable.

Return type float

spikes_per_second
The assumed maximum spikes per second of an incoming population. Used when calculating the ring buffer weight scaling. Settable.

Return type float

synapse_dynamics
The synapse dynamics used by the synapses e.g. plastic or static. Settable.

Return type *AbstractSynapseDynamics* or None

vertex_executable_suffix
The suffix of the executable name due to the type of synapses in use.

Return type str

write_data_spec(*spec*, *application_vertex*, *post_vertex_slice*, *machine_vertex*, *machine_graph*, *application_graph*, *routing_info*, *weight_scale*, *machine_time_step*)

Parameters

- **spec** (*DataSpecificationGenerator*) – The data specification to write to
- **application_vertex** (AbstractPopulationVertex) – The vertex owning the synapses
- **post_vertex_slice** (*Slice*) – The part of the vertex we're dealing with
- **machine_vertex** (PopulationMachineVertex) – The machine vertex
- **machine_graph** (*MachineGraph*) – The graph containing the machine vertex
- **application_graph** (*ApplicationGraph*) – The graph containing the application vertex
- **routing_info** (*RoutingInfo*) – How messages are routed
- **weight_scale** (*float*) – How to scale the weights of the synapses
- **machine_time_step** (*float*) –

spynnaker.pyNN.models.populations package

Module contents

A population is a group of neurons with the same neuron model and synaptic model, but possibly (usually!) varying connectivity and configuration parameters.

A population view is a subset of a population, created by slicing the population:

```
view = population[n:m]
```

An assembly is an agglomeration of populations and population views, created by adding them together:

```
assembly = population_1 + population_2
```

Note: sPyNNaker only has incomplete support for assemblies; do not use.

class spynnaker.pyNN.models.populations.**Assembly**(**args*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

A group of neurons, may be heterogeneous, in contrast to a Population where all the neurons are of the same type.

Parameters

- **populations** (Population *or* PopulationView) – the populations or views to form the assembly out of
- **kwargs** – may contain *label* (a string describing the assembly)

class spynnaker.pyNN.models.populations.**IDMixin**(*population*, *id*)
Bases: object

Instead of storing IDs as integers, we store them as ID objects, which allows a syntax like:

```
p[3,4].tau_m = 20.0
```

where p is a Population object.

Parameters

- **population** (Population) –
- **id** (*int*) –

as_view()
Return a PopulationView containing just this cell.

Return type *PopulationView*

celltype

Return type *AbstractPyNNModel*

get_initial_value(*variable*)
Get the initial value of a state variable of the cell.

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

Return type float

get_parameters()
Return a dict of all cell parameters.

Return type dict(str, ..)

id

Return type int

initialize(***initial_values*)
Set the initial value of a state variable of the cell.

inject(*current_source*)
Inject current from a current source object into the cell.

Parameters **current_source** (*NeuronCurrentSource*) –

is_standard_cell

Return type bool

local
Whether this cell is local to the current MPI node.

Return type bool

position
Return the cell position in 3D space. Cell positions are stored in an array in the parent Population, if any, or within the ID object otherwise. Positions are generated the first time they are requested and then cached.

Return type ndarray

record(*variables*, *to_file=None*, *sampling_interval=None*)
Record the given variable(s) of this cell.

Parameters

- **variables** (*str or list(str)*) – either a single variable name or a list of variable names. For a given celltype class, celltype.recordable contains a list of variables that can be recorded for that celltype.
- **to_file** (*neo.io.baseio.BaseIO or str*) – If specified, should be a Neo IO instance and write_data() will be automatically called when end() is called.
- **sampling_interval** (*int*) – should be a value in milliseconds, and an integer multiple of the simulation timestep.

set_initial_value(*variable*, *value*)
Set the initial value of a state variable of the cell. :param str variable: The name of the variable :param float value: The value of the variable

set_parameters(***parameters*)
Set cell parameters, given as a sequence of parameter=value arguments.

class spynnaker.pyNN.models.populations.**Population**(*size*, *cellclass*, *cellparams=None*, *structure=None*, *initial_values=None*, *label=None*, *constraints=None*, *additional_parameters=None*)
Bases: `spynnaker.pyNN.models.populations.population_base.PopulationBase`

PyNN 0.9 population object.

Parameters

- **size** (*int*) – The number of neurons in the population
- **cellclass** (*type or AbstractPyNNModel*) – The implementation of the individual neurons.
- **cellparams** (*dict(str,object) or None*) – Parameters to pass to `cellclass` if it is a class to instantiate. Must be `None` if `cellclass` is an instantiated object.
- **structure** (*BaseStructure*) –
- **initial_values** (*dict(str,float)*) – Initial values of state variables
- **label** (*str*) – A label for the population
- **constraints** (*list(AbstractConstraint)*) – Any constraints on how the population is deployed to SpiNNaker.
- **additional_parameters** (*dict(str, ..)*) – Additional parameters to pass to the vertex creation function.

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 MPI nodes.

Return type iterable(*IDMixin*)

all_cells

Return type list(*IDMixin*)

annotations
The annotations given by the end user

Return type dict(str, ..)

can_record(*variable*)
Determine whether *variable* can be recorded from this population.

Parameters **variable** (*str*) – The variable to answer the question about

Return type bool

celltype
Implements the PyNN expected celltype property

Returns The celltype this property has been set to

Return type *AbstractPyNNModel*

conductance_based
True if the population uses conductance inputs

Return type bool

static create(*cellclass*, *cellparams=None*, *n=1*)
Pass through method to the constructor defined by PyNN. Create `n` cells all of the same type.

Parameters

- **cellclass** (*type or AbstractPyNNModel*) – see `__init__()`
- **cellparams** (*dict(str, object) or None*) – see `__init__()`
- **n** (*int*) – see `__init__()` (`size` parameter)

Returns A New Population

Return type *Population*

describe(*template='population_default.txt'*, *engine='default'*)
Returns a human-readable description of the population.

The output may be customized 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.

Parameters

- **template** (*str*) – Template filename
- **engine** (*str or TemplateEngine or None*) – Template substitution engine

Return type str or dict

find_units (*variable*)
Get the units of a variable

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

Returns The units of the variable

Return type str

first_id
The ID of the first member of the population.

Return type int

get (*parameter_names*, *gather=True*, *simplify=True*)
Get the values of a parameter for every local cell in the population.

Parameters

- **parameter_names** (*str or iterable(str)*) – Name of parameter. This is either a single string or a list of strings
- **gather** (*bool*) – pointless on sPyNNaker
- **simplify** (*bool*) – ignored

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_data (*variables='all'*, *gather=True*, *clear=False*, *annotations=None*)
Return a Neo Block containing the data (spikes, state variables) recorded from the Assembly.

Parameters

- **variables** (*str or list(str)*) – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** (*bool*) – Whether to collect data from all MPI nodes or just the current node.

  Note: This is irrelevant on sPyNNaker, which always behaves as if this parameter is True.

- **clear** (*bool*) – Whether recorded data will be deleted from the `Assembly`.
- **annotations** (*dict(str, ..)*) – annotations to put on the neo block

Return type Block

Raises **ConfigurationException** – If the variable or variables have not been previously set to record.

get_data_by_indexes (*variables*, *indexes*, *clear=False*, *annotations=None*)
Return a Neo *Block* containing the data (spikes, state variables) recorded from the Assembly.

Parameters

- **variables** (*str or list(str)*) – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.

- **indexes** (*list (int)*) – List of neuron indexes to include in the data. Clearly only neurons recording will actually have any data. If None will be taken as all recording as in `get_data()`
- **clear** (*bool*) – Whether recorded data will be deleted.
- **annotations** (*dict (str, ..)*) – annotations to put on the neo block

Return type Block

Raises **ConfigurationException** – If the variable or variables have not been previously set to record.

get_initial_value(*variable*, *selector=None*)
Deprecated since version 6.0: Use `initial_values()` instead.

get_spike_counts(*gather=True*)
Return the number of spikes for each neuron.

Return type ndarray

id_to_index(*id*)
Given the ID(s) of cell(s) in the Population, return its (their) index (order in the Population).

Defined by http://neuralensemble.org/docs/PyNN/reference/populations.html

Parameters **id** (*int or iterable (int)*) –

Return type int or iterable(int)

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.

Defined by http://neuralensemble.org/docs/PyNN/reference/populations.html

Parameters **cell_id** (*int or iterable (int)*) –

Return type int or iterable(int)

index_to_id(*index*)
Given the index (order in the Population) of cell(s) in the Population, return their ID(s)

Parameters **index** (*int or iterable (int)*) –

Return type int or iterable(int)

initial_values

Return type dict

initialize(***kwargs*)
Set initial values of state variables, e.g. the membrane potential. Values passed to `initialize()` may be:

- single numeric values (all neurons set to the same value), or
- `RandomDistribution` objects, or
- lists / arrays of numbers of the same size as the population mapping functions, where a mapping function accepts a single argument (the cell index) and returns a single number.

Values should be expressed in the standard PyNN units (i.e. millivolts, nanoamps, milliseconds, microsiemens, nanofarads, event per second).

Examples:

```
p.initialize(v=-70.0)
p.initialize(v=rand_distr, gsyn_exc=0.0)
p.initialize(v=lambda i: -65 + i / 10.0)
```

inject (*current_source*)
Connect a current source to all cells in the Population.

Defined by http://neuralensemble.org/docs/PyNN/reference/populations.html

label
The label of the population

Return type str

last_id
The ID of the last member of the population.

Return type int

local_size
The number of local cells

Defined by http://neuralensemble.org/docs/PyNN/reference/populations.html

mark_no_changes ()
Mark this population as not having changes to be mapped.

position_generator

Return type callable((int), ndarray)

positions
Return the position array for structured populations.

Returns a 2D array, one row per cell. Each row is three long, for X,Y,Z

Return type ndarray

record (*variables*, *to_file=None*, *sampling_interval=None*)
Record the specified variable or variables for all cells in the Population or view.

Parameters

- **variables** (*str or list(str)*) – either a single variable name or a list of variable names. For a given celltype class, `celltype.recordable` contains a list of variables that can be recorded for that celltype.
- **to_file** (*io or rawio or str*) – a file to automatically record to (optional). `write_data()` will be automatically called when *sim.end()* is called.
- **sampling_interval** (*int*) – a value in milliseconds, and an integer multiple of the simulation timestep.

requires_mapping
Whether this population requires mapping.

Return type bool

sample (*n*, *rng=None*)
Randomly sample *n* cells from the Population, and return a PopulationView object.

Parameters

- **n** (*int*) – The number of cells to put in the view.
- **rng** (*NumpyRNG*) – The random number generator to use

Return type *PopulationView*

set (***parameters*)
Set parameters of this population.

Parameters **parameters** – The parameters 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_by_selector(1, "tau_m", 20.0).
p.set_by_selector(1, {'tau_m':20, 'v_rest':-65})
```

Parameters

- **selector** – See `RangedList.set_value_by_selector()` as this is just a pass through method
- **parameter** (*str or dict(str, int or float or list(int) or list(float))*) – the parameter to set or dictionary of parameters to set
- **value** (*int or float or list(int) or list(float)*) – 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.

Parameters **constraint** (*AbstractConstraint*) –

set_initial_value (*variable*, *value*, *selector=None*)
Deprecated since version 6.0: Use *initialize()* instead.

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** (*int*) – the new value for the max atoms per core.

size
The number of neurons in the population

Return type int

spinnaker_get_data (*variable*)
Public accessor for getting data as a numpy array, instead of the neo based object

Parameters **variable** (*str or list(str)*) – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.

Returns array of the data

Return type ndarray

structure
Return the structure for the population.

Return type BaseStructure or None

tset (***kwargs*)
Deprecated since version 5.0: Use `set(parametername=value_array)` instead.

write_data (*io*, *variables='all'*, *gather=True*, *clear=False*, *annotations=None*)
Write recorded data to file, using one of the file formats supported by Neo.

Parameters

- **io** (*neo.io.baseio.BaseIO or str*) – a Neo IO instance, or a string for where to put a neo instance
- **variables** (*str or list(str)*) – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** (*bool*) – Whether to bring all relevant data together.

  Note: SpiNNaker always gathers.

- **clear** (*bool*) – clears the storage data if set to true after reading it back
- **annotations** (*dict(str, ..)*) – annotations to put on the neo block

Raises **ConfigurationException** – If the variable or variables have not been previously set to record.

class spynnaker.pyNN.models.populations.**PopulationBase**
Bases: `object`

Shared methods between *Population*s and *PopulationView*s.

Mainly pass through and not implemented.

all_cells
An array containing the cell IDs of all neurons in the Population (all MPI nodes).

Return type list(int)

getSpikes (**args*, ***kwargs*)
Deprecated since version 5.0: Use `get_data('spikes')` instead.

get_data (*variables='all'*, *gather=True*, *clear=False*, *annotations=None*)
Return a Neo Block containing the data(spikes, state variables) recorded from the Population.

Parameters

- **variables** (*str or list(str)*) – Either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** (*bool*) – For parallel simulators, if this is True, all data will be gathered to all nodes and the Neo Block will contain data from all nodes. Otherwise, the Neo Block will contain only data from the cells simulated on the local node.

  Note: SpiNNaker always gathers.

- **clear** (*bool*) – If this is True, recorded data will be deleted from the Population.
- **annotations** (*None or dict(str, ..)*) – annotations to put on the neo block

Return type Block

get_gsyn (**args*, ***kwargs*)
Deprecated since version 5.0: Use `get_data(['gsyn_exc', 'gsyn_inh'])` instead.

get_spike_counts (*gather=True*)
Returns a dict containing the number of spikes for each neuron.

The dict keys are neuron IDs, not indices.

Parameters **gather** (*bool*) – For parallel simulators, if this is True, all data will be gathered to all nodes and the Neo Block will contain data from all nodes. Otherwise, the Neo Block will contain only data from the cells simulated on the local node.

Note: SpiNNaker always gathers.

Return type dict(int, int)

get_v (**args*, ***kwargs*)
Deprecated since version 5.0: Use `get_data('v')` instead.

inject (*current_source*)
Connect a current source to all cells in the Population.

Warning: Currently unimplemented.

Parameters **current_source** (*pyNN.neuron.standardmodels.electrodes.NeuronCurrentSource*) –

is_local (*id*)
Indicates whether the cell with the given ID exists on the local MPI node.

Return type bool

local_cells
An array containing the cell IDs of those neurons in the Population that exist on the local MPI node.

Return type list(int)

local_size
Return the number of cells in the population on the local MPI node.

Return type int

meanSpikeCount (**args*, ***kwargs*)
Deprecated since version 5.0: Use `mean_spike_count()` instead.

mean_spike_count (*gather=True*)
Returns the mean number of spikes per neuron.

Parameters **gather** (*bool*) – For parallel simulators, if this is True, all data will be gathered to all nodes and the Neo Block will contain data from all nodes. Otherwise, the Neo Block will contain only data from the cells simulated on the local node.

Note: SpiNNaker always gathers.

Return type float

nearest (*position*)
Return the neuron closest to the specified position.

Warning: Currently unimplemented.

position_generator

Note: NO PyNN description of this method.

Warning: Currently unimplemented.

positions

Note: NO PyNN description of this method.

Warning: Currently unimplemented.

Return type ndarray(tuple(float, float, float))

printSpikes (*filename*, *gather=True*)
Deprecated since version 5.0: Use `write_data(file, 'spikes')` instead.

Note: Method signature is the PyNN0.7 one

print_gsyn (*filename*, *gather=True*)
Deprecated since version 5.0: Use `write_data(file, ['gsyn_exc', 'gsyn_inh'])` instead.

Note: Method signature is the PyNN0.7 one

print_v (*filename*, *gather=True*)
Deprecated since version 5.0: Use `write_data(file, 'v')` instead.

Note: Method signature is the PyNN0.7 one

receptor_types ()

Note: NO PyNN description of this method.

Warning: Currently unimplemented.

record (*variables*, *to_file=None*, *sampling_interval=None*)
Record the specified variable or variables for all cells in the Population or view.

Parameters

- **variables** (`str or list(str)`) – either a single variable name or a list of variable names. For a given celltype class, *celltype.recordable* contains a list of variables that can be recorded for that celltype.
- **to_file** (`io or rawio or str`) – a file to automatically record to (optional). *write_data()* will be automatically called when *end()* is called.
- **sampling_interval** (`int`) – a value in milliseconds, and an integer multiple of the simulation timestep.

record_gsyn (*sampling_interval=1*, *to_file=None*)
Deprecated since version 5.0: Use `record(['gsyn_exc', 'gsyn_inh'])` instead.

Note: Method signature is the PyNN 0.7 one with the extra non-PyNN *sampling_interval* and *indexes*

record_v (*sampling_interval=1*, *to_file=None*)
Deprecated since version 5.0: Use `record('v')` instead.

Note: Method signature is the PyNN 0.7 one with the extra non-PyNN *sampling_interval* and *indexes*

rset (**args*, ***kwargs*)
Deprecated since version 5.0: Use `set(parametername=rand_distr)` instead.

save_positions (*file*)
Save positions to file. The output format is index x y z

Warning: Currently unimplemented.

structure
The spatial structure of the parent Population.

Warning: Currently unimplemented.

Return type BaseStructure

tset (***kwargs*)
Deprecated since version 5.0: Use `set(parametername=value_array)` instead.

write_data (*io*, *variables='all'*, *gather=True*, *clear=False*, *annotations=None*)
Write recorded data to file, using one of the file formats supported by Neo.

Parameters

- **io** (*io or rawio or str*) – a Neo IO instance, or a string for where to put a Neo instance
- **variables** (*str or list(str)*) – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** (*bool*) – For parallel simulators, if this is True, all data will be gathered to all nodes and the Neo Block will contain data from all nodes. Otherwise, the Neo Block will contain only data from the cells simulated on the local node. This is pointless on sPyNNaker.

  Note: SpiNNaker always gathers.

- **clear** (*bool*) – clears the storage data if set to true after reading it back
- **annotations** (*None or dict(str, ..)*) – annotations to put on the Neo block

class `spynnaker.pyNN.models.populations.`**PopulationView**(*parent*, *selector*, *label=None*)

Bases: `spynnaker.pyNN.models.populations.population_base.PopulationBase`

A view of a subset of neurons within a *Population*.

In most ways, Populations and PopulationViews have the same behaviour, i.e., they can be recorded, connected with Projections, etc. It should be noted that any changes to neurons in a PopulationView will be reflected in the parent Population and *vice versa.*

It is possible to have views of views.

Note: Selector to Id is actually handled by `AbstractSized`.

Parameters

- **parent** (`Population or PopulationView`) – the population or view to make the view from
- **selector** (*None or slice or int or list(bool) or list(int) or ndarray(bool) or ndarray(int)*) – a slice or numpy mask array. The mask array should either be a boolean array (ideally) of the same size as the parent, or an integer array containing cell indices, i.e. if *p.size == 5* then:

```
PopulationView(p, array([False, False, True, False, True]))
PopulationView(p, array([2, 4]))
PopulationView(p, slice(2, 5, 2))
```

  will all create the same view.

- **label** (*str*) – A label for the view

all()

Iterator over cell IDs (on all MPI nodes).

Return type iterable(IDMixin)

all_cells

An array containing the cell IDs of all neurons in the Population (all MPI nodes).

Return type list(*IDMixin*)

can_record(*variable*)

Determine whether variable can be recorded from this population.

Return type bool

celltype

The type of neurons making up the underlying Population.

Return type *AbstractPyNNModel*

conductance_based

Indicates whether the post-synaptic response is modelled as a change in conductance or a change in current.

Return type bool

describe(*template='populationview_default.txt'*, *engine='default'*)

Returns a human-readable description of the population view.

The output may be customized 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.

Parameters

- **template** (*str*) – Template filename
- **engine** (*str or TemplateEngine or None*) – Template substitution engine

Return type str or dict

find_units(*variable*)

Get the units of a variable

Warning: No PyNN description of this method.

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

Returns The units of the variable

Return type str

get(*parameter_names*, *gather=False*, *simplify=True*)

Get the values of the given parameters for every local cell in the population, or, if `gather=True`, for all cells in the population.

Values will be expressed in the standard PyNN units (i.e. millivolts, nanoamps, milliseconds, microsiemens, nanofarads, event per second).

Note: SpiNNaker always gathers.

Parameters

- **parameter_names** (*str or list(str)*) –
- **gather** (*bool*) –
- **simplify** (*bool*) –

Return type iterable(float)

get_data(*variables='all'*, *gather=True*, *clear=False*, *annotations=None*)
Return a Neo Block containing the data(spikes, state variables) recorded from the Population.

Parameters

- **variables** (*str or list(str)*) – Either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** (*bool*) – For parallel simulators, if gather is True, all data will be gathered to all nodes and the Neo Block will contain data from all nodes. Otherwise, the Neo Block will contain only data from the cells simulated on the local node.

Note: SpiNNaker always gathers.

- **clear** (*bool*) – If True, recorded data will be deleted from the Population.
- **annotations** (*dict(str, ..)*) – annotations to put on the neo block

Return type Block

Raises **ConfigurationException** – If the variable or variables have not been previously set to record.

get_spike_counts(*gather=True*)
Returns a dict containing the number of spikes for each neuron.

The dict keys are neuron IDs, not indices.

Note: Implementation of this method is different to Population as the Populations uses PyNN 7 version of the `get_spikes` method which does not support indexes.

Parameters **gather** (*bool*) –

Note: SpiNNaker always gathers.

Return type dict(int,int)

grandparent
Returns the parent Population at the root of the tree (since the immediate parent may itself be a PopulationView).

The name "grandparent" is of course a little misleading, as it could be just the parent, or the great, great, great, . . . , grandparent.

Return type *Population*

id_to_index(*id*)
Given the ID(s) of cell(s) in the PopulationView, return its / their index / indices(order in the PopulationView).

assert pv.id_to_index(pv[3]) == 3

Parameters **id** (*int or list(int)*) –

Return type int or list(int)

index_in_grandparent (*indices*)
Given an array of indices, return the indices in the parent population at the root of the tree.

Parameters **indices** (*list (int)*) –

Return type list(int)

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

Return type dict(str, ..)

initialize (***initial_values*)
Set initial values of state variables, e.g. the membrane potential. Values passed to `initialize()` may be:

- single numeric values (all neurons set to the same value), or
- `RandomDistribution` objects, or
- lists / arrays of numbers of the same size as the population mapping functions, where a mapping function accepts a single argument (the cell index) and returns a single number.

Values should be expressed in the standard PyNN units (i.e. millivolts, nanoamps, milliseconds, microsiemens, nanofarads, events per second).

Examples:

```
p.initialize(v=-70.0)
p.initialize(v=rand_distr, gsyn_exc=0.0)
p.initialize(v=lambda i: -65 + i / 10.0)
```

label
A label for the Population View.

Return type str

mask
The selector mask that was used to create this view.

Return type None or slice or int or list(bool) or list(int) or ndarray(bool) or ndarray(int)

parent
A reference to the parent Population (that this is a view of).

Return type *Population*

record (*variables*, *to_file=None*, *sampling_interval=None*)
Record the specified variable or variables for all cells in the Population or view.

Parameters

- **variables** (*str or list (str)*) – either a single variable name, or a list of variable names, or `all` to record everything. For a given celltype class, celltype.recordable contains a list of variables that can be recorded for that celltype.
- **to_file** (*io or rawio or str*) – If specified, should be a Neo IO instance and `write_data()` will be automatically called when *sim.end()* is called.
- **sampling_interval** (*int*) – should be a value in milliseconds, and an integer multiple of the simulation timestep.

sample (*n*, *rng=None*)
Randomly sample *n* cells from the Population view, and return a new PopulationView object.

Parameters

- **n** (*int*) – The number of cells to select
- **rng** (*NumpyRNG*) – Random number generator

Return type *PopulationView*

set (***parameters*)
Set one or more parameters for every cell in the population. Values passed to *set()* may be:

- single values,
- `RandomDistribution` objects, or
- lists / arrays of values of the same size as the population mapping functions, where a mapping function accepts a single argument (the cell index) and returns a single value.

Here, a "single value" may be either a single number or a list / array of numbers (e.g. for spike times).

Values should be expressed in the standard PyNN units (i.e. millivolts, nanoamps, milliseconds, microsiemens, nanofarads, event per second).

Examples:

```
p.set(tau_m=20.0, v_rest=-65).
p.set(spike_times=[0.3, 0.7, 0.9, 1.4])
p.set(cm=rand_distr, tau_m=lambda i: 10 + i / 10.0)
```

size
The total number of neurons in the Population View.

Return type int

write_data (*io*, *variables='all'*, *gather=True*, *clear=False*, *annotations=None*)
Write recorded data to file, using one of the file formats supported by Neo.

Parameters

- **io** (*neo.io.BaseIO or str*) – a Neo IO instance or the name of a file to write
- **variables** (*str or list(str)*) – either a single variable name or a list of variable names. These must have been previously recorded, otherwise an Exception will be raised.
- **gather** (*bool*) – For parallel simulators, if this is True, all data will be gathered to the master node and a single output file created there. Otherwise, a file will be written on each node, containing only data from the cells simulated on that node.

  Note: SpiNNaker always gathers.

- **clear** (*bool*) – If this is True, recorded data will be deleted from the Population.
- **annotations** (*dict(str, ..)*) – should be a dict containing simple data types such as numbers and strings. The contents will be written into the output data file as metadata.

Raises `ConfigurationException` – If the variable or variables have not been previously set to record.

spynnaker.pyNN.models.spike_source package

Submodules

spynnaker.pyNN.models.spike_source.spike_source_array_vertex module

class spynnaker.pyNN.models.spike_source.spike_source_array_vertex.**SpikeSourceArrayVertex**(*n*
s
c
s
l
l
n
n
s
t

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*)

Clear the recorded data from the object

Parameters

- **buffer_manager** (*BufferManager*) – the buffer manager object
- **placements** (*Placements*) – the placements 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*, *buffer_manager*, *machine_time_step*)

Get the recorded spikes from the object

Parameters

- **placements** (*Placements*) – the placements object
- **buffer_manager** (*BufferManager*) – the buffer manager object
- **machine_time_step** (*int*) – the time step of the simulation, in microseconds

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

Return type ndarray(tuple(int,int))

get_spikes_sampling_interval()

Return the current sampling interval for spikes

Returns Sampling interval in microseconds

Return type float

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** (*int or None*) – The interval at which spikes are recorded. Must be a whole multiple of the timestep. None will be taken as the timestep.
- **indexes** (*list(int) or None*) – The indexes of the neurons that will record spikes. If None the assumption is all neurons are recording

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

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

Parameters

- **selector** (*None or slice or int or list(bool) or list(int)*) – See RangedList.set_value_by_selector as this is just a pass through method
- **key** (*str*) – the name of the parameter to change
- **value** (*float or int or list(float) or list(int)*) – the new value of the parameter to assign

spike_times

The spike times of the spike source array

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_supports_database_injection.AbstractSupportsDatabaseInjection, spinn_front_end_common.interface.profiling.abstract_has_profile_data.AbstractHasProfileData, spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, spinn_front_end_common.abstract_models.abstract_rewrites_data_specification.AbstractRewritesDataSpecification, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification, spynnaker.pyNN.models.abstract_models.abstract_read_parameters_before_set.AbstractReadParametersBeforeSet

class EXTRA_PROVENANCE_DATA_ENTRIES

Bases: enum.Enum

Entries for the provenance data generated by standard neuron models.

TDMA_MISSED_SLOTS = 0

The number of pre-synaptic events

FAST_RATE_PER_TICK_CUTOFF = 10

PARAMS_BASE_WORDS = 13

class POISSON_SPIKE_SOURCE_REGIONS

Bases: enum.Enum

An enumeration.

POISSON_PARAMS_REGION = 1

PROFILER_REGION = 5

PROVENANCE_REGION = 4

RATES_REGION = 2

SPIKE_HISTORY_REGION = 3

SYSTEM_REGION = 0

TDMA_REGION = 6

PROFILE_TAG_LABELS = {0: 'TIMER', 1: 'PROB_FUNC'}

SEED_OFFSET_BYTES = 36

SEED_SIZE_BYTES = 16

SLOW_RATE_PER_TICK_CUTOFF = 0.01

generate_data_specification (*spec*, *placement*, *machine_time_step*, *time_scale_factor*, *routing_info*, *data_n_time_steps*, *graph*, *first_machine_time_step*)

Generate a data specification.

Parameters

- **spec** (*DataSpecificationGenerator*) – The data specification to write to
- **placement** (*Placement*) – The placement the vertex is located at
- **machine_time_step** (*int*) –
- **time_scale_factor** (*int*) –
- **routing_info** (*RoutingInfo*) –
- **data_n_time_steps** (*int*) –
- **graph** (*MachineGraph*) –
- **first_machine_time_step** (*int*) –

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_profile_data (*transceiver*, *placement*)

Get the profile data recorded during simulation

Parameters

- **transceiver** (*Transceiver*) –
- **placement** (*Placement*) –

Return type 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_in_injection_mode
Whether this vertex is actually in injection mode.

Return type bool

max_spikes_per_second()
Get maximum expected number of spikes per second

Parameters variable (*str*) – the variable to find units from

Returns the units as a string.

Return type str

max_spikes_per_ts(*machine_time_step*)
Get maximum expected number of spikes per timestep

Parameters machine_time_step (*int*) – The timestime used in ms

Return type int

poisson_param_region_address(*placement*, *transceiver*)

poisson_rate_region_address(*placement*, *transceiver*)

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

Parameters

- **transceiver** (*Transceiver*) – the SpinnMan interface
- **placement** (*Placement*) – the placement of a vertex
- **vertex_slice** (*Slice*) – the slice of atoms for this vertex

Return type None

regenerate_data_specification(*spec*, *placement*, *machine_time_step*, *routing_info*, *graph*, *first_machine_time_step*)
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?
- **machine_time_step** (*int*) –
- **routing_info** (*RoutingInfo*) –
- **graph** (*MachineGraph*) –
- **first_machine_time_step** (*int*) –

reload_required(*first_machine_time_step*)
Return true if any data region needs to be reloaded

Return type bool

reserve_memory_regions(*spec*, *placement*)
Reserve memory regions for Poisson source parameters and output buffer.

Parameters

- **spec** (*DataSpecificationGenerator*) – the data specification writer
- **placement** (*Placement*) – the location this vertex resides on in the machine

Returns None

resources_required
The resources required by the vertex

Return type ResourceContainer

set_reload_required(*new_value*)
Indicate that the regions have been reloaded

Parameters **new_value** – the new value

Return type None

spynnaker.pyNN.models.spike_source.spike_source_poisson_machine_vertex.**get_rates_bytes**(*verte*
*rate_*
Gets the size of the Poisson rates in bytes

Parameters **vertex_slice** (*Slice*) –

Return type int

spynnaker.pyNN.models.spike_source.spike_source_poisson_vertex module

class spynnaker.pyNN.models.spike_source.spike_source_poisson_vertex.**SpikeSourcePoissonVert**

Bases: spinn_front_end_common.abstract_models.impl.tdma_aware_application_vertex.TDMAAwareApplicationVertex, 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.common.simple_population_settable.SimplePopulationSettable, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl, pacman.model.partitioner_interfaces.legacy_partitioner_api.LegacyPartitionerAPI

A Poisson Spike source object

Parameters

- **n_neurons** (*int*) –
- **constraints** (*iterable(AbstractConstraint)*) –
- **label** (*str*) –
- **seed** (*float*) –
- **max_atoms_per_core** (*int*) –
- **model** (SpikeSourcePoisson) –
- **rate** (*iterable(float)*) –
- **start** (*iterable(int)*) –
- **duration** (*iterable(int)*) –
- **splitter** (*AbstractSplitterCommon*) –

SPIKE_RECORDING_REGION_ID = 0

clear_spike_recording(*buffer_manager*, *placements*)

Clear the recorded data from the object

Parameters

- **buffer_manager** (*BufferManager*) – the buffer manager object
- **placements** (*Placements*) – the placements object

Return type None

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.

Returns The created machine vertex

Return type MachineVertex

describe()

Return 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.

Return type dict(str, ..)

duration

durations

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 partition 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_recording_sdram_usage(*vertex_slice*, *machine_time_step*)

Parameters

- **vertex_slice** (*Slice*) –
- **machine_time_step** (*int*) –

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
- **vertex_slice** –
- **machine_time_step** (*int*) –

Returns a resource container that contains a `CPUCyclesPerTickResource`, `DTCMResource` and `SDRAMResource`

Return type ResourceContainer

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

Parameters

- **placements** (*Placements*) – the placements object
- **buffer_manager** (*BufferManager*) – the buffer manager object
- **machine_time_step** (*int*) – the time step of the simulation, in microseconds

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

Return type ndarray(tuple(int,int))

get_spikes_sampling_interval()
Return the current sampling interval for spikes

Returns Sampling interval in microseconds

Return type float

is_recording_spikes()
Determine if spikes are being recorded

Returns True if spikes are being recorded, False otherwise

Return type bool

kiss_seed(*vertex_slice*)

mark_no_changes()
Marks the point after which changes are reported, so that new changes can be detected before the next check.

max_rate

max_spikes_per_ts(*machine_time_step*)

Parameters **machine_time_step** (*int*) –

n_atoms
The number of atoms in the vertex

Return type int

n_profile_samples

rate

rate_change

rates

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

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** (*int or None*) – The interval at which spikes are recorded. Must be a whole multiple of the timestep. None will be taken as the timestep.
- **indexes** (*list(int) or None*) – 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** (*str*) – the name of the parameter to change
- **value** (*Any or float or int or list(float) or list(int)*) – the new value of the parameter to assign

start

starts

time_to_spike

update_kiss_seed (*vertex_slice*, *seed*)
updates a kiss seed from the machine

Parameters

- **vertex_slice** – the vertex slice to update seed of
- **seed** – the seed

Return type None

Module contents

class spynnaker.pyNN.models.spike_source.**SpikeSourceArray** (*spike_times=None*)
Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex (*n_neurons*, *label*, *constraints*, *splitter*)
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 (AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type ApplicationVertex

default_population_parameters = {'splitter': None}

class spynnaker.pyNN.models.spike_source.**SpikeSourceArrayVertex** (*n_neurons*, *spike_times*, *constraints*, *label*, *max_atoms_per_core*, *model*, *splitter*)
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*)
Clear the recorded data from the object

Parameters

- **buffer_manager** (*BufferManager*) – the buffer manager object
- **placements** (*Placements*) – the placements 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*, *buffer_manager*, *machine_time_step*)
Get the recorded spikes from the object

Parameters

- **placements** (*Placements*) – the placements object
- **buffer_manager** (*BufferManager*) – the buffer manager object
- **machine_time_step** (*int*) – the time step of the simulation, in microseconds

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

Return type ndarray(tuple(int,int))

get_spikes_sampling_interval()
Return the current sampling interval for spikes

Returns Sampling interval in microseconds

Return type float

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** (*int or None*) – The interval at which spikes are recorded. Must be a whole multiple of the timestep. None will be taken as the timestep.
- **indexes** (*list(int) or None*) – The indexes of the neurons that will record spikes. If None the assumption is all neurons are recording

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

Parameters

- **selector** (*None or slice or int or list(bool) or list(int)*) – See RangedList.set_value_by_selector as this is just a pass through method
- **key** (*str*) – the name of the parameter to change
- **value** (*float or int or list(float) or list(int)*) – the new value of the parameter to assign

spike_times
The spike times of the spike source array

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, splitter*)
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(AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type ApplicationVertex

default_population_parameters = {'max_rate': None, 'seed': None, 'splitter': 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

class spynnaker.pyNN.models.spike_source.**SpikeSourcePoissonMachineVertex**(*resources_required, is_recording, constraints=None, label=None, app_vertex=None, vertex_slice=None*)

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_supports_database_injection.AbstractSupportsDatabaseInjection`, `spinn_front_end_common.interface.profiling.abstract_has_profile_data.AbstractHasProfileData`, `spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary`, `spinn_front_end_common.abstract_models.abstract_rewrites_data_specification.AbstractRewritesDataSpecification`, `spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification`, `spynnaker.pyNN.models.abstract_models.abstract_read_parameters_before_set.AbstractReadParametersBeforeSet`

class EXTRA_PROVENANCE_DATA_ENTRIES

Bases: `enum.Enum`

Entries for the provenance data generated by standard neuron models.

TDMA_MISSED_SLOTS = 0

The number of pre-synaptic events

FAST_RATE_PER_TICK_CUTOFF = 10

PARAMS_BASE_WORDS = 13

class POISSON_SPIKE_SOURCE_REGIONS

Bases: `enum.Enum`

An enumeration.

POISSON_PARAMS_REGION = 1

PROFILER_REGION = 5

PROVENANCE_REGION = 4

RATES_REGION = 2

SPIKE_HISTORY_REGION = 3

SYSTEM_REGION = 0

TDMA_REGION = 6

PROFILE_TAG_LABELS = {0: 'TIMER', 1: 'PROB_FUNC'}

SEED_OFFSET_BYTES = 36

SEED_SIZE_BYTES = 16

SLOW_RATE_PER_TICK_CUTOFF = 0.01

generate_data_specification(*spec*, *placement*, *machine_time_step*, *time_scale_factor*, *routing_info*, *data_n_time_steps*, *graph*, *first_machine_time_step*)

Generate a data specification.

Parameters

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

- **time_scale_factor** (*int*) –
- **routing_info** (*RoutingInfo*) –
- **data_n_time_steps** (*int*) –
- **graph** (*MachineGraph*) –
- **first_machine_time_step** (*int*) –

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_profile_data(*transceiver*, *placement*)
Get the profile data recorded during simulation

Parameters

- **transceiver** (*Transceiver*) –
- **placement** (*Placement*) –

Return type 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_in_injection_mode
Whether this vertex is actually in injection mode.

Return type bool

max_spikes_per_second()
Get maximum expected number of spikes per second

Parameters **variable** (*str*) – the variable to find units from

Returns the units as a string.

Return type str

max_spikes_per_ts(*machine_time_step*)
Get maximum expected number of spikes per timestep

Parameters **machine_time_step** (*int*) – The timestime used in ms

Return type int

poisson_param_region_address(*placement*, *transceiver*)

poisson_rate_region_address(*placement*, *transceiver*)

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

Parameters

- **transceiver** (*Transceiver*) – the SpinnMan interface
- **placement** (*Placement*) – the placement of a vertex
- **vertex_slice** (*Slice*) – the slice of atoms for this vertex

Return type None

regenerate_data_specification(*spec*, *placement*, *machine_time_step*, *routing_info*, *graph*, *first_machine_time_step*)
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?
- **machine_time_step** (*int*) –
- **routing_info** (*RoutingInfo*) –
- **graph** (*MachineGraph*) –
- **first_machine_time_step** (*int*) –

reload_required(*first_machine_time_step*)
Return true if any data region needs to be reloaded

Return type bool

reserve_memory_regions(*spec*, *placement*)
Reserve memory regions for Poisson source parameters and output buffer.

Parameters

- **spec** (*DataSpecificationGenerator*) – the data specification writer
- **placement** (*Placement*) – the location this vertex resides on in the machine

Returns None

resources_required
The resources required by the vertex

Return type ResourceContainer

set_reload_required(*new_value*)
Indicate that the regions have been reloaded

Parameters **new_value** – the new value

Return type None

class spynnaker.pyNN.models.spike_source.**SpikeSourcePoissonVariable**(*rates*, *starts*, *durations=None*)
Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex(*n_neurons*, *label*, *constraints*, *seed*, *splitter*)
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 (AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type ApplicationVertex

default_population_parameters = {'seed': None, 'splitter': 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

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.

Parameters

- **n_delay_stages** (*int*) –

- **delay_per_stage** (*int*) –
- **vertex_slice** (*Slice*) –

add_delay (*source_id*, *stage*)

Parameters

- **source_id** (*int*) –
- **stage** (*int*) –

delay_block

Return type ndarray

class spynnaker.pyNN.models.utility_models.delays.**DelayExtensionMachineVertex** (*resources_required*, *label*, *constraints=None*, *app_vertex=None*, *vertex_slice=None*)

Bases: pacman.model.graphs.machine.machine_vertex.MachineVertex, spinn_front_end_common.interface.provenance.provides_provenance_data_from_machine_impl.ProvidesProvenanceDataFromMachineImpl, spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification

Parameters

- **resources_required** (*ResourceContainer*) – The resources required by the vertex
- **label** (*str*) – The optional name of the vertex
- **constraints** (*iterable(AbstractConstraint)*) – The optional initial constraints of the vertex
- **app_vertex** (*ApplicationVertex*) – The application vertex that caused this machine vertex to be created. If None, there is no such application vertex.
- **vertex_slice** (*Slice*) – The slice of the application vertex that this machine vertex implements.

BACKGROUND_MAX_QUEUED_NAME = 'Max_backgrounds_queued'

BACKGROUND_OVERLOADS_NAME = 'Times_the_background_queue_overloaded'

COUNT_SATURATION_ERROR_MESSAGE = 'The delay extension {} has dropped {} packets becaus

COUNT_SATURATION_NAME = 'saturation_count'

DELAYED_FOR_TRAFFIC_NAME = 'Number_of_times_delayed_to_spread_traffic'

class EXTRA_PROVENANCE_DATA_ENTRIES

Bases: enum.Enum

An enumeration.

MAX_BACKGROUND_QUEUED = 11

N_BACKGROUND_OVERLOADS = 12

N_BUFFER_OVERFLOWS = 4

N_DELAYS = 5

N_LATE_SPIKES = 10

N_PACKETS_ADDED = 2

N_PACKETS_DROPPED_DUE_TO_INVALID_KEY = 9

N_PACKETS_LOST_DUE_TO_COUNT_SATURATION = 7

N_PACKETS_PROCESSED = 1

N_PACKETS_RECEIVED = 0

N_PACKETS_SENT = 3

N_PACKETS_WITH_INVALID_NEURON_IDS = 8

N_TIMES_TDMA_FELL_BEHIND = 6

INPUT_BUFFER_LOST_ERROR_MESSAGE = 'The input buffer for {} on {}, {}, {} lost packets

INPUT_BUFFER_LOST_NAME = 'Times_the_input_buffer_lost_packets'

INVALID_KEY_COUNT_NAME = 'invalid_key_count'

INVALID_NEURON_IDS_ERROR_MESSAGE = 'The delay extension {} has dropped {} packets beca

INVALID_NEURON_ID_COUNT_NAME = 'invalid_neuron_count'

MISMATCH_ADDED_FROM_PROCESSED_ERROR_MESSAGE = 'The delay extension {} on {}, {}, {} on

MISMATCH_ADDED_FROM_PROCESSED_NAME = 'Number_of_packets_added_to_delay_slot'

MISMATCH_PROCESSED_FROM_RECEIVED_ERROR_MESSAGE = 'The delay extension {} on {}, {}, {}

N_EXTRA_PROVENANCE_DATA_ENTRIES = 13

N_LATE_SPIKES_MESSAGE_DROP = '{} packets from {} on {}, {}, {} were dropped from the i

N_LATE_SPIKES_MESSAGE_NO_DROP = '{} packets from {} on {}, {}, {} arrived too late to

N_LATE_SPIKES_NAME = 'Number_of_late_spikes'

N_PACKETS_PROCESSED_NAME = 'Number_of_packets_processed'

N_PACKETS_RECEIVED_NAME = 'Number_of_packets_received'

N_PACKETS_SENT_NAME = 'Number_of_packets_sent'

PACKETS_DROPPED_FROM_INVALID_KEY_ERROR_MESSAGE = 'The delay extension {} has dropped {

gen_on_machine()

Determine if the given slice needs to be generated on the machine

Parameters **vertex_slice** (*Slice*) –

Return type bool

generate_data_specification(*spec*, *placement*, *machine_graph*, *routing_infos*, *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

- **machine_graph** (*MachineGraph*) –
- **routing_infos** (*RoutingInfo*) –
- **machine_time_step** (*int*) – machine time step of the sim.
- **time_scale_factor** (*int*) – the time scale factor of the sim.

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_n_keys_for_partition(*_partition*)
Get the number of keys required by the given partition of edges.

Parameters **_partition** (*OutgoingEdgePartition*) – An partition that comes out of this vertex

Returns The number of keys required

Return type int

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

write_delay_parameters(*spec*, *vertex_slice*, *key*, *incoming_key*, *incoming_mask*)
Generate Delay Parameter data

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **vertex_slice** (*Slice*) –
- **key** (*int*) –
- **incoming_key** (*int*) –
- **incoming_mask** (*int*) –

class spynnaker.pyNN.models.utility_models.delays.**DelayExtensionVertex**(*n_neurons*, *delay_per_stage*, *max_delay_to_support*, *source_vertex*, *constraints=None*, *label='DelayExtension'*)

Bases: spinn_front_end_common.abstract_models.impl.tdma_aware_application_vertex.TDMAAwareApplicationVertex, spynnaker.pyNN.models.abstract_models.abstract_has_delay_stages.AbstractHasDelayStages, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.AbstractProvidesOutgoingPartitionConstraints

Provide delays to incoming spikes in multiples of the maximum delays of a neuron (typically 16 or 32)

Parameters

- **n_neurons** (*int*) – the number of neurons
- **delay_per_stage** (*int*) – the delay per stage
- **max_delay_to_support** (*int*) – the max delay this will cover
- **source_vertex** (*ApplicationVertex*) – where messages are coming from
- **constraints** (*iterable(AbstractConstraint)*) – the vertex constraints
- **label** (*str*) – the vertex label

MAX_DTCM_AVAILABLE = 54756

MAX_TICKS_POSSIBLE_TO_SUPPORT = 128

MISMATCHED_DELAY_PER_STAGE_ERROR_MESSAGE = 'The delay per stage is already set to {}, a

SAFETY_FACTOR = 5000

add_delays(*vertex_slice*, *source_ids*, *stages*)

Add delayed connections for a given vertex slice

Parameters

- **vertex_slice** (*Slice*) –
- **source_ids** (*list(int)*) –
- **stages** (*list(int)*) –

add_generator_data(*max_row_n_synapses*, *max_delayed_row_n_synapses*, *pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_information*, *max_stage*, *max_delay_per_stage*, *machine_time_step*)

Add delays for a connection to be generated

Parameters

- **max_row_n_synapses** (*int*) – The maximum number of synapses in a row
- **max_delayed_row_n_synapses** (*int*) – The maximum number of synapses in a delay row
- **pre_slices** (*list(Slice)*) – The list of slices of the pre application vertex
- **post_slices** (*list(Slice)*) – The list of slices of the post application vertex

- **pre_vertex_slice** (*Slice*) – The slice of the pre applcation vertex currently being considered
- **post_vertex_slice** (*Slice*) – The slice of the post application vertex currently being considered
- **synapse_information** (SynapseInformation) – The synapse information of the connection
- **synapse_information** –
- **max_stage** (*int*) – The maximum delay stage
- **machine_time_step** (*int*) – sim machine time step
- **max_delay_per_stage** (*int*) – max delay per stage

delay_blocks_for (*vertex_slice*)

delay_generator_data (*vertex_slice*)

delay_per_stage

drop_late_spikes

gen_on_machine (*vertex_slice*)
Determine if the given slice needs to be generated on the machine

Parameters **vertex_slice** (*Slice*) –

Return type bool

static get_max_delay_ticks_supported (*delay_ticks_at_post_vertex*)

get_outgoing_partition_constraints (*partition*)
Get constraints to be added to the given edge partition 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)

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

Return type int The maximum number of delay stages required by any connection out of this delay extension vertex

Return type int

set_new_n_delay_stages_and_delay_per_stage (*new_post_vertex_n_delay*, *new_max_delay*)

source_vertex

Return type ApplicationVertex

spynnaker.pyNN.models.utility_models.spike_injector package

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*, *splitter*)
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(AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type ApplicationVertex

default_population_parameters = {'port': None, 'reserve_reverse_ip_tag': False, 'spl

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(AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type 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, Any)

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** (*iterable(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** (*iterable(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*)
Get an argument initialisation dictionary by examining an `__init__` method or function.

Parameters

- **init** (*callable*) – The method.
- **skip** (*frozenset(str)*) – The arguments to be skipped, if any
- **include** (*frozenset(str)*) – The arguments that must be present, if any

Returns an initialisation dictionary

Return type dict(str, Any)

spynnaker.pyNN.models.projection module

class spynnaker.pyNN.models.projection.**Projection**(*pre_synaptic_population*, *post_synaptic_population*, *connector*, *synapse_type=None*, *source=None*, *receptor_type=None*, *space=None*, *label=None*)

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.

Parameters

- **pre_synaptic_population** (PopulationBase) –
- **post_synaptic_population** (PopulationBase) –
- **connector** (AbstractConnector) –
- **synapse_type** (AbstractSynapseDynamics) –
- **source** (*None*) – Unsupported; must be None
- **receptor_type** (*str*) –
- **space** (*Space*) –
- **label** (*str*) –

get(*attribute_names*, *format*, *gather=True*, *with_address=True*, *multiple_synapses='last'*)

Get a parameter/attribute of the projection.

Note: SpiNNaker always gathers.

Parameters

- **attribute_names** (*str or iterable(str)*) – list of attributes to gather
- **format** (*str*) – "list" or "array"
- **gather** (*bool*) – gather over all nodes
- **with_address** (*bool*) – True if the source and target are to be included
- **multiple_synapses** (*str*) – What to do with the data if format="array" and if the multiple source-target pairs with the same values exist. Currently only "last" is supported

Returns values selected

getDelays(*format='list'*, *gather=True*)

Deprecated since version 5.0: Use `get('delay')` instead.

getSynapseDynamics(*parameter_name*, *format='list'*, *gather=True*)

Deprecated since version 5.0: Use `get(parameter_name)` instead.

getWeights(*format='list'*, *gather=True*)

Deprecated since version 5.0: Use `get('weight')` instead.

label

Return type str

mark_no_changes()
Mark this projection as not having changes to be mapped.

post
The post-population or population view.

Return type *PopulationBase*

pre
The pre-population or population view.

Return type *PopulationBase*

printDelays(*file*, *format='list'*, *gather=True*)
Deprecated since version 5.0: Use `save('delay')` instead.

Print synaptic weights to file. In the array format, zeros are printed for non-existent connections.

printWeights(*file*, *format='list'*, *gather=True*)
Deprecated since version 5.0: Use `save('weight')` instead.

requires_mapping
Whether this projection requires mapping.

Return type bool

save(*attribute_names*, *file*, *format='list'*, *gather=True*, *with_address=True*)
Print synaptic attributes (weights, delays, etc.) to file. In the array format, zeros are printed for non-existent connections. Values will be expressed in the standard PyNN units (i.e., millivolts, nanoamps, milliseconds, microsiemens, nanofarads, event per second).

Note: SpiNNaker always gathers.

Parameters

- **attribute_names** (*str or list(str)*) –
- **file** (*str or pyNN.recording.files.BaseFile*) – filename or open handle (which will be closed)
- **format** (*str*) –
- **gather** (*bool*) – Ignored
- **with_address** (*bool*) –

saveConnections(*file*, *gather=True*, *compatible_output=True*)
Deprecated since version 5.0: Use `save('all')` instead.

set(***attributes*)

Warning: Not implemented.

size(*gather=True*)
Return the total number of connections.

Note: SpiNNaker always gathers.

> **Warning:** Not implemented.

Parameters **gather** (*bool*) – If False, only get the number of connections locally.

weightHistogram(*min=None*, *max=None*, *nbins=10*)
Deprecated since version 5.0: Use `numpy.histogram` on the weights instead.

Return a histogram of synaptic weights. If `min` and `max` are not given, the minimum and maximum weights are calculated automatically.

spynnaker.pyNN.models.recorder module

class spynnaker.pyNN.models.recorder.**Recorder**(*population*, *vertex*)
Bases: `object`

Object to hold recording behaviour, used by populations.

Parameters

- **population** (`Population`) – the population to record for
- **vertex** (*ApplicationVertex*) – the SpiNNaker graph vertex used by the population

cache_data()
Store data for later extraction

extract_neo_block(*variables*, *view_indexes*, *clear*, *annotations*)
Extracts block from the vertices and puts them into a Neo block

Parameters

- **variables** (*list (str)*) – the variables to extract
- **view_indexes** (*slice*) – the indexes to be included in the view
- **clear** (*bool*) – if the variables should be cleared after reading
- **annotations** (*dict (str, object)*) – annotations to put on the Neo block

Returns The Neo block

Return type Block

get_all_possible_recordable_variables()
All variables that could be recorded.

Return type set(str)

get_all_recording_variables()
All variables that have been set to record.

Return type set(str)

get_recorded_matrix(*variable*)
Perform safety checks and get the recorded data from the vertex in matrix format.

Parameters **variable** (*str*) – The variable name to read. Supported variable names are: `gsyn_exc`, `gsyn_inh`, `v`

Returns data, indexes, sampling_interval

Return type tuple(ndarray, list(int), float)

get_recorded_pynn7 (*variable*)
Get recorded data in PyNN 0.7 format. Must not be spikes.

Parameters **variable** (*str*) – The name of the variable to get. Supported variable names are: `gsyn_exc`, `gsyn_inh`, `v`

Return type ndarray

get_spikes ()
How to get spikes (of a population's neurons) from the recorder.

Returns the spikes (event times) from the underlying vertex

Return type ndarray

record (*variables*, *to_file*, *sampling_interval*, *indexes*)
Same as record but without non-standard PyNN warning

This method is non-standard PyNN and is intended only to be called by record in a Population, View or Assembly

Parameters

- **variables** (*str or list(str) or None*) – either a single variable name or a list of variable names. For a given celltype class, `celltype.recordable` contains a list of variables that can be recorded for that celltype. Can also be `None` to reset the list of variables.
- **to_file** (*io or rawio or str*) – a file to automatically record to (optional). `write_data()` will be automatically called when *sim.end()* is called.
- **sampling_interval** (*int*) – a value in milliseconds, and an integer multiple of the simulation timestep.
- **indexes** (*None or list(int)*) – The indexes of neurons to record from. This is non-standard PyNN and equivalent to creating a view with these indexes and asking the View to record.

turn_off_all_recording (*indexes=None*)
Turns off recording, is used by a pop saying `.record()`

Parameters **indexes** (*list or None*) –

turn_on_record (*variable*, *sampling_interval=None*, *to_file=None*, *indexes=None*)
Tell the vertex to record data.

Parameters

- **variable** (*str*) – The variable to record, supported variables to record are: `gsyn_exc`, `gsyn_inh`, `v`, `spikes`.
- **sampling_interval** (*int*) – the interval to record them
- **to_file** (*neo.io.baseio.BaseIO or str or None*) – If set, a file to write to (by handle or name)
- **indexes** (*list(int) or None*) – List of indexes to record or None for all

write_to_files_indicators
What variables should be written to files, and where should they be written.

Return type dict(str, neo.io.baseio.BaseIO or str or None)

Module contents

spynnaker.pyNN.protocols package

Module contents

class spynnaker.pyNN.protocols.**MunichIoEthernetProtocol**

Bases: `object`

Implementation of the Munich robot IO protocol, communicating over ethernet.

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** (*int or None*) – The optional instance key to use
- **uart_id** (*int*) – The ID of the UART when needed

class MODES

Bases: `enum.Enum`

types of modes supported by this protocol

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*, *time=None*)

generic_motor_total_period_key

instance_key

The key of this instance of the protocol

Return type int

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

Return type *MODES*

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** (`RetinaKey`) – the new key for the retina
- **retina_payload** (`RetinaPayload or None`) – the new payload for the set retina key command packet
- **time** (`int or float or None`) – when to transmit this packet

Returns the command to send

Return type MultiCastCommand

set_retina_transmission_key

turn_off_sensor_reporting(*sensor_id, time=None*)

turn_off_sensor_reporting_key

uart_id

Return type int

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

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsBinomialImpl**
Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

An implementation of AbstractRandomStats for binomial distributions

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsExponentialImpl**
Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

An implementation of AbstractRandomStats for exponential distributions

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsGammaImpl**
Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

An implementation of AbstractRandomStats for gamma distributions

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsLogNormalImpl**
Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

An implementation of AbstractRandomStats for log normal distributions

cdf(*dist*, *v*)
Return the cumulative distribution function value for the value v

high (*dist*)
Return the variance of the distribution

low (*dist*)
Return the variance of the distribution

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsNormalClippedImpl**
Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

An implementation of AbstractRandomStats for normal distributions that are clipped to a boundary (redrawn)

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsNormalImpl**
Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

An implementation of AbstractRandomStats for normal distributions

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsPoissonImpl**
Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

An implementation of AbstractRandomStats for poisson distributions

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsRandIntImpl**
Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

An implementation of AbstractRandomStats for uniform distributions

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsScipyImpl**(*distribution_type*)

Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

A Random Statistics object that uses scipy directly

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsUniformImpl**

Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

An implementation of AbstractRandomStats for uniform distributions

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

class spynnaker.pyNN.utilities.random_stats.**RandomStatsVonmisesImpl**

Bases: spynnaker.pyNN.utilities.random_stats.abstract_random_stats.AbstractRandomStats

An implementation of AbstractRandomStats for vonmises distributions

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

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** (*int*) – 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

Return type *SpynnakerRangedList*

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** (*int or None*) – Fixed length of the list; if None, the value must be a sized object.
- **value** (*object or Sized*) – 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** (*bool*) – 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.bit_field_utilities module

spynnaker.pyNN.utilities.bit_field_utilities.**ELEMENTS_USED_IN_BIT_FIELD_HEADER = 2**

n_filters, pointer for array

spynnaker.pyNN.utilities.bit_field_utilities.**ELEMENTS_USED_IN_EACH_BIT_FIELD = 3**

number of elements

spynnaker.pyNN.utilities.bit_field_utilities.**N_ELEMENTS_IN_EACH_KEY_N_ATOM_MAP = 2**

n elements in each key to n atoms map for bitfield (key, n atoms)

spynnaker.pyNN.utilities.bit_field_utilities.**N_KEYS_DATA_SET_IN_WORDS = 1**

n key to n neurons maps size in words

spynnaker.pyNN.utilities.bit_field_utilities.**N_REGIONS_ADDRESSES = 6**

the regions addresses needed (pop table, synaptic matrix, direct matrix, bit_field, bit field builder, bit_field_key, structural region)

spynnaker.pyNN.utilities.bit_field_utilities.**exact_sdram_for_bit_field_builder_region**()

Gets the SDRAM requirement for the builder region

Returns the SDRAM requirement for the builder region

Return type int

spynnaker.pyNN.utilities.bit_field_utilities.**get_estimated_sdram_for_bit_field_region**(*app_g ver- tex*)

estimates the SDRAM for the bit field region

Parameters

- **app_graph** (*ApplicationGraph*) – the app graph
- **vertex** (*ApplicationVertex*) – app vertex

Returns the estimated number of bytes used by the bit field region

Return type int

`spynnaker.pyNN.utilities.bit_field_utilities.`**`get_estimated_sdram_for_key_region`**(*app_graph*, *vertex*)

gets an estimate of the bitfield builder region

Parameters

- **app_graph** (*ApplicationGraph*) – the app graph
- **vertex** (*ApplicationVertex*) – app vertex

Returns SDRAM needed

Return type int

`spynnaker.pyNN.utilities.bit_field_utilities.`**`reserve_bit_field_regions`**(*spec*, *machine_graph*, *n_key_map*, *vertex*, *bit_field_builder_region*, *bit_filter_region*, *bit_field_key_region*)

reserves the regions for the bitfields

Parameters

- **spec** (*DataSpecificationGenerator*) – dsg spec writer
- **machine_graph** (*MachineGraph*) – machine graph
- **n_key_map** (*AbstractMachinePartitionNKeysMap*) – map between partitions and n keys
- **vertex** (*MachineVertex*) – machine vertex
- **bit_field_builder_region** (*int*) – region id for the builder region
- **bit_filter_region** (*int*) – region id for the bitfield region
- **bit_field_key_region** (*int*) – region id for the key map

spynnaker.pyNN.utilities.bit_field_utilities.**write_bitfield_init_data**(*spec*, *machine_vertex*, *machine_graph*, *routing_info*, *n_key_map*, *bit_field_builder_region*, *master_pop_region_id*, *synaptic_matrix_region_id*, *direct_matrix_region_id*, *bit_field_region_id*, *bit_field_key_map_region_id*, *structural_dynamics_region_id*, *has_structural_dynamics_regi*

writes the init data needed for the bitfield generator

Parameters

- **spec** (*DataSpecificationGenerator*) – data spec writer
- **machine_vertex** (*MachineVertex*) – machine vertex
- **machine_graph** (*MachineGraph*) – machine graph
- **routing_info** (*RoutingInfo*) – keys
- **n_key_map** (*AbstractMachinePartitionNKeysMap*) – map for edge to n keys
- **bit_field_builder_region** (*int*) – the region id for the bitfield builder
- **master_pop_region_id** (*int*) – the region id for the master pop table
- **synaptic_matrix_region_id** (*int*) – the region id for the synaptic matrix
- **direct_matrix_region_id** (*int*) – the region id for the direct matrix
- **bit_field_region_id** (*int*) – the region id for the bit-fields
- **bit_field_key_map_region_id** (*int*) – the region id for the key map
- **structural_dynamics_region_id** (*int*) – the region id for the structural
- **has_structural_dynamics_region** (*bool*) – whether the core has a structural_dynamics region

spynnaker.pyNN.utilities.constants module

spynnaker.pyNN.utilities.constants.**LIVE_POISSON_CONTROL_PARTITION_ID = 'CONTROL'**

The partition ID used for Poisson live control data

spynnaker.pyNN.utilities.constants.**MIN_SUPPORTED_DELAY = 1**

the minimum supported delay slot between two neurons

spynnaker.pyNN.utilities.constants.**OUT_SPIKE_BYTES = 32**

The number of bytes for each spike line

spynnaker.pyNN.utilities.constants.**OUT_SPIKE_SIZE = 8**
The size of each output spike line

class spynnaker.pyNN.utilities.constants.**POPULATION_BASED_REGIONS**
Bases: enum.Enum

Regions for populations.

BIT_FIELD_BUILDER = 13

BIT_FIELD_FILTER = 12

BIT_FIELD_KEY_MAP = 14

CONNECTOR_BUILDER = 10

DIRECT_MATRIX = 11

NEURON_PARAMS = 1

NEURON_RECORDING = 7

POPULATION_TABLE = 3

PROFILING = 9

PROVENANCE_DATA = 8

STRUCTURAL_DYNAMICS = 6

SYNAPSE_DYNAMICS = 5

SYNAPSE_PARAMS = 2

SYNAPTIC_MATRIX = 4

SYSTEM = 0

spynnaker.pyNN.utilities.constants.**POP_TABLE_MAX_ROW_LENGTH = 256**
The maximum row length of the master population table

spynnaker.pyNN.utilities.constants.**SPIKE_PARTITION_ID = 'SPIKE'**
The partition ID used for spike data

spynnaker.pyNN.utilities.constants.**SYNAPTIC_ROW_HEADER_WORDS = 3**
Words: 2 for row length and number of rows and 1 for plastic region size (which might be 0)

spynnaker.pyNN.utilities.data_cache module

class spynnaker.pyNN.utilities.data_cache.**DataCache**(*label*, *description*, *segment_number*, *recording_start_time*, *t*)
Bases: object

Storage object to hold all the data to (re)create a Neo Segment

Note: Required because deep-copy does not work on neo Objects

Stores the Data shared by all variable types at the top level and holds a cache for the variable specific data.

Parameters

- **label** (*str*) – cache label

- **description** (*str or dict*) – cache description
- **segment_number** (*int*) – cache segment number
- **recording_start_time** (*float*) – when this cache was started in recording space.
- **t** (*float*) – time

description

get_data (*variable*)
Get the variable cache for the named variable

Parameters **variable** (*str*) – name of variable to get cache for

Returns The cache data, IDs, indexes and units

Return type *VariableCache*

has_data (*variable*)
Checks if data for a variable has been cached

Parameters **variable** (*str*) – Name of variable

Returns True if there is cached data

Return type bool

label

rec_datetime

recording_start_time

save_data (*variable*, *data*, *indexes*, *n_neurons*, *units*, *sampling_interval*)
Saves the data for one variable in this segment

Parameters

- **variable** (*str*) – name of variable data applies to
- **data** (*ndarray*) – raw data in sPyNNaker format
- **indexes** (*ndarray*) – population indexes for which data should be returned
- **n_neurons** (*int*) – Number of neurons in the population, regardless of if they where recording or not.
- **units** (*str*) – the units in which the data is
- **sampling_interval** (*float or int*) – The number of milliseconds between samples.

segment_number

t

variables
Provides a list of which variables data has been cached for

Return type Iterator (str)

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.

get (*projection*, *attribute*)
Allow getting data from a given projection and attribute

Parameters

- **projection** (`Projection`) – the projection data was extracted from
- **attribute** (*list(int) or tuple(int) or None*) – the attribute to retrieve

Returns the attribute data in a connection holder

Return type *ConnectionHolder*

set (*projection*, *attribute*, *data*)
Allow the addition of data from a projection and attribute.

Parameters

- **projection** (`Projection`) – the projection data was extracted from
- **attribute** (*list(int) or tuple(int) or None*) – the attribute to store
- **data** (`ConnectionHolder`) – 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.neo_compare module

spynnaker.pyNN.utilities.neo_compare.**compare_analogsignal** (*as1*, *as2*, *same_length=True*)
Compares two analogsignal Objects to see if they are the same

Parameters

- **as1** (*AnalogSignal*) – first analogsignal holding list of individual analogsignal Objects
- **as2** (*AnalogSignal*) – second analogsignal holding list of individual analogsignal Objects
- **same_length** (*bool*) – Flag to indicate if the same length of data is held, i.e., all spikes up to the same time. If False allows one trains to have additional data after the first ends. This is used to compare data extracted part way with data extracted at the end.

Raises **AssertionError** – If the analogsignals are not equal spynnaker.pyNN.utilities.neo_compare.**compare_blocks**(*neo1*, *neo2*, *same_runs=True*, *same_data=True*, *same_length=True*)

Compares two neo Blocks to see if they hold the same data.

Parameters

- **neo1** (*Block*) – First block to check
- **neo2** (*Block*) – Second block to check
- **same_runs** (*bool*) – Flag to signal if blocks are the same length. If False extra segments in the larger block are ignored
- **same_data** (*bool*) – Flag to indicate if the same type of data is held, i.e., same spikes, v, gsyn_exc and gsyn_inh. If False only data in both blocks is compared
- **same_length** (*bool*) – Flag to indicate if the same length of data is held, i.e., all spikes up to the same time. If False allows one trains to have additional data after the first ends. This is used to compare data extracted part way with data extracted at the end.

Raises **AssertionError** – If the blocks are not equal

spynnaker.pyNN.utilities.neo_compare.**compare_segments**(*seg1*, *seg2*, *same_data=True*, *same_length=True*)

Parameters

- **seg1** (*Segment*) – First Segment to check
- **seg2** (*Segment*) – Second Segment to check
- **same_data** (*bool*) – Flag to indicate if the same type of data is held, i.e., same spikes, v, gsyn_exc and gsyn_inh. If False only data in both blocks is compared
- **same_length** (*bool*) – Flag to indicate if the same length of data is held, i.e., all spikes up to the same time. If False allows one trains to have additional data after the first ends. This is used to compare data extracted part way with data extracted at the end.

Raises **AssertionError** – If the segments are not equal

spynnaker.pyNN.utilities.neo_compare.**compare_spiketrain**(*spiketrain1*, *spiketrain2*, *same_length=True*)

Checks two Spiketrains have the exact same data

Parameters

- **spiketrain1** (*SpikeTrain*) – first spiketrain
- **spiketrain2** (*SpikeTrain*) – second spiketrain
- **same_length** (*bool*) – Flag to indicate if the same length of data is held, i.e., all spikes up to the same time. If False allows one trains to have additional spikes after the first ends. This is used to compare data extracted part way with data extracted at the end.

Return type None

Raises **AssertionError** – If the spiketrains are not equal

spynnaker.pyNN.utilities.neo_compare.**compare_spiketrains**(*spiketrains1*, *spiketrains2*, *same_data=True*, *same_length=True*)

Check two Lists of SpikeTrains have the exact same data

Parameters

- **spiketrains1** (*list (SpikeTrain)*) – First list SpikeTrains to compare
- **spiketrains2** (*list (SpikeTrain)*) – Second list of SpikeTrains to compare
- **same_data** (*bool*) – Flag to indicate if the same type of data is held, i.e., same spikes, v, gsyn_exc and gsyn_inh. If False allows one or both lists to be Empty. Even if False none empty lists must be the same length
- **same_length** (*bool*) – Flag to indicate if the same length of data is held, i.e., all spikes up to the same time. If False allows one trains to have additional spikes after the first ends. This is used to compare data extracted part way with data extracted at the end.

Raises **AssertionError** – If the spiketrains are not equal

spynnaker.pyNN.utilities.neo_convertor module

spynnaker.pyNN.utilities.neo_convertor.**convert_analog_signal**(*signal_array*, *time_unit=<sphinx.ext.autodoc.importer._M object>*)

Converts part of a NEO object into told spynnaker7 format

Parameters

- **signal_array** (*AnalogSignal*) – Extended Quantities object
- **time_unit** (*quantities.unitquantity.UnitTime*) – Data time unit for time index

Return type ndarray

spynnaker.pyNN.utilities.neo_convertor.**convert_data**(*data*, *name*, *run=0*)

Converts the data into a numpy array in the format ID, time, value

Parameters

- **data** (*Block*) – Data as returned by a getData() call
- **name** (*str*) – Name of the data to be extracted. Same values as used in getData()
- **run** (*int*) – Zero based index of the run to extract data for

Return type ndarray

spynnaker.pyNN.utilities.neo_convertor.**convert_data_list**(*data*, *name*, *runs=None*)

Converts the data into a list of numpy arrays in the format ID, time, value

Parameters

- **data** (*Block*) – Data as returned by a getData() call
- **name** (*str*) – Name of the data to be extracted. Same values as used in getData()
- **runs** (*list (int) or None*) – List of Zero based index of the run to extract data for. Or None to extract all runs

Return type list(ndarray)

spynnaker.pyNN.utilities.neo_convertor.**convert_gsyn**(*gsyn_exc*, *gsyn_inh*)

Converts two neo objects into the spynnaker7 format

Note: It is acceptable for both neo parameters to be the same object

Parameters

- **gsyn_exc** (*Block*) – neo with gsyn_exc data
- **gsyn_inh** (*Block*) – neo with gsyn_exc data

Return type ndarray

`spynnaker.pyNN.utilities.neo_convertor.`**`convert_gsyn_exc_list`**(*data*, *runs=None*)

Converts the gsyn_exc into a list numpy array one per segment (all runs) in the format ID, time, value

Parameters

- **data** (*Block*) – The data to convert; it must have Gsyn_exc data in it
- **runs** (*list(int) or None*) – List of Zero based index of the run to extract data for. Or None to extract all runs

Return type list(ndarray)

`spynnaker.pyNN.utilities.neo_convertor.`**`convert_gsyn_inh_list`**(*data*, *runs=None*)

Converts the gsyn_inh into a list numpy array one per segment (all runs) in the format ID, time, value

Parameters

- **data** (*Block*) – The data to convert; it must have Gsyn_inh data in it
- **runs** (*list(int) or None*) – List of Zero based index of the run to extract data for. Or None to extract all runs

Return type list(ndarray)

`spynnaker.pyNN.utilities.neo_convertor.`**`convert_spikes`**(*neo*, *run=0*)

Extracts the spikes for run one from a Neo Object

Parameters

- **neo** (*Block*) – neo Object including Spike Data
- **run** (*int*) – Zero based index of the run to extract data for

Return type ndarray

`spynnaker.pyNN.utilities.neo_convertor.`**`convert_spiketrains`**(*spiketrains*)

Converts a list of spiketrains into spynnaker7 format

Parameters **spiketrains** (*list(SpikeTrain)*) – List of SpikeTrains

Return type ndarray

`spynnaker.pyNN.utilities.neo_convertor.`**`convert_v_list`**(*data*, *runs=None*)

Converts the voltage into a list numpy array one per segment (all runs) in the format ID, time, value

Parameters

- **data** (*Block*) – The data to convert; it must have V data in it
- **runs** (*list(int) or None*) – List of Zero based index of the run to extract data for. Or None to extract all runs

Return type list(ndarray)

`spynnaker.pyNN.utilities.neo_convertor.`**`count_spikes`**(*neo*)

Help function to count the number of spikes in a list of spiketrains

Only counts run 0

Parameters **neo** (*Block*) – Neo Object which has spikes in it

Returns The number of spikes in the first segment

`spynnaker.pyNN.utilities.neo_convertor.`**`count_spiketrains`**(*spiketrains*)
Help function to count the number of spikes in a list of spiketrains

Parameters **spiketrains** (*list (SpikeTrain)*) – List of SpikeTrains

Returns Total number of spikes in all the spiketrains

Return type int

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*)
Adds an item to the running statistics.

Parameters **x** (*int or float*) – The item to add

`add_items`(*mean*, *variance*, *n_items*)
Add a bunch of items (via their statistics).

Parameters

- **mean** (*float*) – The mean of the items to add.
- **variance** (*float*) – The variance of the items to add.
- **n_items** (*int*) – The number of items represented.

`mean`
The mean of the items seen.

Return type float

`n_items`
The number of items seen.

Return type int

`standard_deviation`
The population standard deviation of the items seen.

Return type float

`variance`
The variance of the items seen.

Return type float

spynnaker.pyNN.utilities.spynnaker_failed_state module

class `spynnaker.pyNN.utilities.spynnaker_failed_state.`**`SpynnakerFailedState`**(*name*)
Bases: *spynnaker.pyNN.spynnaker_simulator_interface.SpynnakerSimulatorInterface*, `spinn_front_end_common.utilities.failed_state.FailedState`

Marks the simulation as failed.

config
Provides access to the configuration for front end interfaces.

Return type ConfigHandler

dt
The timestep, in milliseconds.

get_current_time()

has_reset_last

max_delay

min_delay

mpi_rank
The MPI rank of the controller node.

name
The name of the simulator. Used to ensure PyNN recording neo blocks are correctly labelled.

num_processes
The number of MPI worker processes.

recorders
The recorders, used by the PyNN state object.

static reset(*annotations=None*)

segment_counter
The number of the current recording segment being generated.

set_number_of_neurons_per_core(*neuron_type*, *max_permitted*)

t
The current simulation time, in milliseconds.

write_on_end

spynnaker.pyNN.utilities.struct module

class spynnaker.pyNN.utilities.struct.**Struct**(*field_types*)
Bases: `object`

Represents a C code structure.

Parameters **field_types** (*list(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(DataType)

get_data(*values*, *offset=0*, *array_size=1*)
Get a numpy array of uint32 of data for the given values

Parameters

- **values** (*list(int or float or list(int) or list(float) or RangedList)*) – A list of values with length the same size as the number of fields returned by field_types

- **offset** (*int*) – The offset into each of the values where to start
- **array_size** (*int*) – The number of structs to generate

Return type ndarray(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** (*int*) – The number of elements in an array of structs

Return type int

numpy_dtype
The numpy data type of the struct

Return type dtype

read_data (*data*, *offset=0*, *array_size=1*)
Read a bytearray of data and convert to struct values

Parameters

- **data** (*bytes or bytearray*) – The data to be read
- **offset** (*int*) – Index of the byte at the start of the valid data
- **array_size** (*int*) – The number of struct elements to read

Returns a list of lists of data values, one list for each struct element

Return type list(float)

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** (*str*) – The file whose parent directory is to be created

spynnaker.pyNN.utilities.utility_calls.**convert_param_to_numpy** (*param*, *no_atoms*)
Convert parameters into numpy arrays.

Parameters

- **param** (*NumpyRNG or int or float or list(int) or list(float) or ndarray*) – the param to convert
- **no_atoms** (*int*) – the number of atoms available for conversion of param

Returns the converted param as an array of floats

Return type ndarray(float)

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** (*DataType*) – The data type to convert to

Returns The converted data as a numpy data type

Return type ndarray(int32)

spynnaker.pyNN.utilities.utility_calls.**create_mars_kiss_seeds**(*rng*, *seed=None*)

generates and checks that the seed values generated by the given random number generator or seed to a random number generator are suitable for use as a mars 64 kiss seed.

Parameters

- **rng** (*None or RandomState*) – the random number generator.
- **seed** (*int or None*) – the seed to create a random number generator if not handed.

Returns a list of 4 ints which are used by the mars64 kiss random number generator for seeds.

Return type list(int)

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

Parameters **n_values** (*int*) – the number of values (starting at 0)

Returns the number of bits required to express that many values

Return type int

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_probable_minimum_selected**(*n_total_trials*, *n_trials*, *selection_prob*, *chance=0.01*)

Get the likely minimum 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.`**`moved_in_v6`**(*old_location*, *new_location*)

Warns the users that they are using an old import.

In version 7 this will ne upgraded to a exception and then later removed

Parameters

- **`old_location`** (*`str`*) – old import
- **`new_location`** (*`str`*) – new import

Raise an exception if in CONTINUOUS_INTEGRATION

`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`** (*`str`*) – absolute path to a file containing the data
- **`min_atom`** (*`int`*) – min neuron ID to which neurons to read in
- **`max_atom`** (*`int`*) – max neuron ID to which neurons to read in
- **`extra`** –
- **`min_time`** (*`float or int`*) – min time slot to read neurons values of.
- **`max_time`** (*`float or int`*) – max time slot to read neurons values of.

Returns a numpy array of (time stamp, atom ID, data value)

Return type ndarray(tuple(float, int, float))

`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 or float`*) – min neuron ID to which neurons to read in
- **`max_atom`** (*`int or float`*) – max neuron ID to which neurons to read in

- **min_time** (*float or int*) – min time slot to read neurons values of.
- **max_time** (*float or 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.ndarray(int, int)

spynnaker.pyNN.utilities.variable_cache module

class spynnaker.pyNN.utilities.variable_cache.**VariableCache**(*data*, *indexes*, *n_neurons*, *units*, *sampling_interval*)

Bases: object

Simple holder method to keep data, IDs, indexes and units together

Typically used to recreate the Neo object for one type of variable for one segment.

Parameters

- **data** (*ndarray*) – raw data in sPyNNaker format
- **indexes** (*list (int)*) – Population indexes for which data was collected
- **n_neurons** (*int*) – Number of neurons in the population, regardless of whether they were recording or not.
- **units** (*str*) – the units in which the data is
- **sampling_interval** (*float or int*) – The number of milliseconds between samples.

data

Return type ndarray

indexes

Return type list(int)

n_neurons

Return type int

sampling_interval

Return type float or int

units

Return type str

Module contents

Submodules

spynnaker.pyNN.abstract_spinnaker_common module

class spynnaker.pyNN.abstract_spinnaker_common.**AbstractSpiNNakerCommon**(*graph_label, database_socket_addresses, n_chips_required, n_boards_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.

Parameters

- **graph_label** (*str*) –
- **database_socket_addresses** (*iterable(SocketAddress)*) –
- **n_chips_required** (*int or None*) –
- **n_boards_required** (*int or None*) –
- **timestep** (*int*) –
- **max_delay** (*float*) –
- **min_delay** (*float*) –
- **hostname** (*str*) –
- **user_extra_algorithm_xml_path** (*str or None*) –
- **user_extra_mapping_inputs** (*dict(str, Any) or None*) –
- **user_extra_algorithms_pre_run** (*list(str) or None*) –
- **time_scale_factor** (*float or None*) –
- **extra_post_run_algorithms** (*list(str) or None*) –
- **extra_mapping_algorithms** (*list(str) or None*) –
- **extra_load_algorithms** (*list(str) or None*) –
- **front_end_versions** (*list(tuple(str,str)) or None*) –

CONFIG_FILE_NAME = 'spynnaker.cfg'

add_application_vertex(*vertex*)

Parameters **vertex** (*ApplicationVertex*) – the vertex to add to the graph

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.

classmethod extended_config_path()

get_projections_data(*projection_to_attribute_map*)

Common data extractor for projection data. Allows fully exploitation of the ????

Parameters **projection_to_attribute_map** (*dict(Projection, list(int) or tuple(int) or None)*) – the projection to attributes mapping

Returns a extracted data object with get method for getting the data

Return type *ExtractedData*

id_counter
The id_counter, currently used by the populations.

Note: Maybe it could live in the pop class???

Return type int

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** (*str*) – absolute search path for binaries

Return type None

reset_number_of_neurons_per_core()

run(*run_time*, *sync_time=0.0*)
Run the model created.

Parameters

- **run_time** (*float or int*) – the time (in milliseconds) to run the simulation for
- **sync_time** (*float*) – If not 0, this specifies that the simulation should pause after this duration. The continue_simulation() method must then be called for the simulation to continue.

Return type None

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 or None*) – 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 or None*) – informs the tool chain if it should turn off the clearing of the routing tables
- **clear_tags** (*bool or None*) – informs the tool chain if it should clear the tags off the machine at stop

Return type None

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.**SpynnakerSplitterConfigurationException**
Bases: spinn_front_end_common.utilities.exceptions.ConfigurationException

Raised when a splitter configuration fails.

exception spynnaker.pyNN.exceptions.**SynapseRowTooBigException**(*max_size*, *message*)
Bases: *spynnaker.pyNN.exceptions.SpynnakerException*

Raised when a synapse row is bigger than is allowed.

Parameters

- **max_size** – the maximum permitted size of row
- **message** – the excepton message

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.

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** (`Population`) – 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** (*EIEIOPrefix*) – if the prefix type is 32 bit or 16 bit
- **message_type** (*EIEIOType*) – If the message is a EIEIO command message, or an EIEIO data message with 16 bit or 32 bit keys.

- **payload_as_time_stamps** (*bool*) –
- **right_shift** (*int*) –
- **use_payload_prefix** (*bool*) –
- **notify** (*bool*) –
- **payload_prefix** (*int or None*) –
- **payload_right_shift** (*int*) –
- **number_of_packets_sent_per_time_step** (*int*) –
- **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
- **label** (*str*) – The label of the gatherer vertex
- **partition_ids** (*list(str)*) – The names of the partitions to create edges for

static activate_live_output_to(*population*, *device*, *partition_id='SPIKE'*)
Activate the output of spikes from a population to an external device. Note that all spikes will be sent to the device.

Parameters

- **population** (*Population*) – The pyNN population object from which spikes will be sent.
- **device** (*Population or ApplicationVertex*) – The pyNN population or external device to which the spikes will be sent.
- **partition_id** (*str*) – The partition ID to activate live output to.

static add_application_vertex(*vertex*)

static add_database_socket_address(*database_notify_host*, *database_notify_port_num*, *database_ack_port_num*)

Parameters

- **database_notify_host** (*str or None*) – Host to talk to tell that the database (and application) is ready.
- **database_notify_port_num** (*int or None*) – Port to talk to tell that the database (and application) is ready.
- **database_ack_port_num** (*int or None*) – Port on which to listen for an acknowledgement that the simulation should start.

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** (*ApplicationVertex*) – the pre-vertex to connect the edge from
- **device_vertex** (*ApplicationVertex*) – the post vertex to connect the edge to
- **partition_id** (*str*) – the partition identifier for making nets

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** (Population) – 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 an external device will receive the database is ready command
- **notify** (*bool*) – adds to the notification protocol if set.
- **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** (*SocketAddress*) – the socket address

static machine_time_step()

static time_scale_factor()

static update_live_packet_gather_tracker(*vertex_to_record_from*, *lpg_label*, *port=None*, *hostname=None*, *board_address=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*, *use_payload_prefix=True*, *payload_prefix=None*, *payload_right_shift=0*, *number_of_packets_sent_per_time_step=0*, *partition_ids=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.

Parameters

- **vertex_to_record_from** (*ApplicationVertex or MachineVertex*) –
- **lpg_label** (*str*) –
- **port** (*int*) –
- **hostname** (*str*) –
- **board_address** (*str*) –
- **tag** (*int*) –
- **strip_sdp** (*bool*) –
- **use_prefix** (*bool*) –
- **key_prefix** (*int*) –
- **prefix_type** (*EIEIOPrefix*) –
- **message_type** (*EIEIOType*) –
- **right_shift** (*int*) –
- **payload_as_time_stamps** (*bool*) –
- **use_payload_prefix** (*bool*) –
- **payload_prefix** (*int*) –
- **payload_right_shift** (*int*) –
- **number_of_packets_sent_per_time_step** (*int*) –
- **partition_ids** (*list (str)*) –

spynnaker.pyNN.spynnaker_simulator_interface module

class spynnaker.pyNN.spynnaker_simulator_interface.**SpynnakerSimulatorInterface**

Bases: spinn_front_end_common.utilities.simulator_interface.SimulatorInterface

The API exposed by the simulator itself.

dt

The timestep, in milliseconds.

get_current_time()

has_reset_last

min_delay

mpi_rank

The MPI rank of the controller node.

name

The name of the simulator. Used to ensure PyNN recording neo blocks are correctly labelled.

num_processes

The number of MPI worker processes.

recorders

The recorders, used by the PyNN state object.

reset (*annotations=None*)

segment_counter
The number of the current recording segment being generated.

set_number_of_neurons_per_core (*neuron_type*, *max_permitted*)

t
The current simulation time, in milliseconds.

Module contents

1.1.2 Submodules

1.1.3 spynnaker.gsyn_tools module

spynnaker.gsyn_tools.**check_gsyn** (*gsyn1*, *gsyn2*)
Compare two arrays of conductances. For testing.

Parameters

- **gsyn1** – An array of conductances.
- **gsyn2** – An array of conductances.

Raises Exception – If the arrays differ.

spynnaker.gsyn_tools.**check_path_gysn** (*path*, *n_neurons*, *runtime*, *gsyn*)
Compare an arrays of conductances with baseline data from a file. For testing.

Parameters

- **path** – A file path.
- **n_neurons** – The number of neurons that produced the data.
- **runtime** – The length of time that the generated data represents.
- **gsyn** – An array of conductances.

Raises Exception – If the arrays differ.

spynnaker.gsyn_tools.**check_sister_gysn** (*sister*, *n_neurons*, *runtime*, *gsyn*)
Compare an arrays of conductances with baseline data from a file next to a specified module. For testing.

Parameters

- **sister** – A module. The file read from will be gsyn.data adjacent to this module.
- **n_neurons** – The number of neurons that produced the data.
- **runtime** – The length of time that the generated data represents.
- **gsyn** – An array of conductances.

Raises Exception – If the arrays differ.

1.1.4 spynnaker.plot_utils module

spynnaker.plot_utils.**heat_plot** (*data_sets*, *ylabel=None*, *title=None*)
Build a heatmap plot or plots.

Parameters

- **data_sets** (*ndarray or list(ndarray)*) – Numpy array of data, or list of numpy arrays of data
- **ylabel** (*str or None*) – The label for the Y axis
- **title** (*str or None*) – The title for the plot

spynnaker.plot_utils.**line_plot**(*data_sets*, *title=None*)
Build a line plot or plots.

Parameters

- **data_sets** (*ndarray or list(ndarray)*) – Numpy array of data, or list of numpy arrays of data
- **title** (*str or None*) – The title for the plot

spynnaker.plot_utils.**plot_spikes**(*spikes*, *title='spikes'*)
Build a spike plot or plots.

Parameters

- **spikes** (*ndarray or list(ndarray)*) – Numpy array of spikes, or list of numpy arrays of spikes
- **title** (*str*) – The title for the plot

1.1.5 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** (*ndarray or list(ndarray)*) – The spikes
- **nNeurons** (*int*) – The number of neurons.
- **lines** (*int*) – Expected number of lines
- **wrap_around** (*bool*) – If True the lines will wrap around when reaching the last neuron.

Raises **Exception** – If there is a problem with the data

spynnaker.spike_checker.**synfire_spike_checker**(*spikes*, *nNeurons*)

Parameters

- **spikes** (*ndarray or list(ndarray)*) – The spike data to check.
- **nNeurons** (*int*) – The number of neurons.

Raises **Exception** – If there is a problem with the data

1.1.6 Module contents

CHAPTER 2

spynnaker8

2.1 spynnaker8 package

2.1.1 Subpackages

2.1.1.1 spynnaker8.external_devices package

Module contents

This contains functions and classes for handling external devices such as the PushBot (http://spinnakermanchester.github.io/docs/push_bot/).

Note: When using external devices, it is normally important to configure your SpiNNaker system to run in real-time mode, which usually reduces numerical accuracy to gain performance.

---

class spynnaker8.external_devices.**EIEIOType**(*value*, *key_bytes*, *payload_bytes*, *doc=''*)
Bases: `enum.Enum`

Possible types of EIEIO packets

KEY_16_BIT = 0

KEY_32_BIT = 2

KEY_PAYLOAD_16_BIT = 1

KEY_PAYLOAD_32_BIT = 3

key_bytes
The number of bytes used by each key element

Return type int

max_value
The maximum value of the key or payload (if there is a payload)

Return type int

payload_bytes
The number of bytes used by each payload element

Return type int

class spynnaker8.external_devices.**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

Parameters

- **n_neurons** (*int*) – Number of neurons
- **spinnaker_link** (*int*) – The SpiNNaker link to which the cochlea is connected
- **label** (*str*) –
- **board_address** (*str*) –

class spynnaker8.external_devices.**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** (*str*) – The retina "mode"
- **retina_key** (*int*) – The value of the top 16-bits of the key
- **spinnaker_link_id** (*int*) – The SpiNNaker link to which the retina is connected
- **polarity** (*str*) – The "polarity" of the retina data
- **label** (*str*) –
- **board_address** (*str*) –

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 partition 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 spynnaker8.external_devices.**MunichRetinaDevice** (*retina_key*, *spinnaker_link_id*, *position*, *label='MunichRetinaDevice'*, *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`

An Omnibot silicon retina device.

Parameters

- **retina_key** (*int*) –
- **spinnaker_link_id** (*int*) – The SpiNNaker link to which the retina is connected
- **position** (*str*) – `LEFT` or `RIGHT`
- **label** (*str*) –
- **polarity** (*str*) – `UP`, `DOWN` or `MERGED`
- **board_address** (*str or None*) –

DOWN_POLARITY = 'DOWN'

LEFT_RETINA = 'LEFT'
Select the left retina

MERGED_POLARITY = 'MERGED'

RIGHT_RETINA = 'RIGHT'
Select the right retina

UP_POLARITY = 'UP'

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

get_outgoing_partition_constraints(*partition*)
Get constraints to be added to the given edge partition 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 spynnaker8.external_devices.**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.abstract.abstract_one_app_one_machine_vertex.AbstractOneAppOneMachineVertex`, `spinn_front_end_common.abstract_models.abstract_vertex_with_dependent_vertices.AbstractVertexWithEdgeToDependentVertices`

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

Parameters

- **spinnaker_link_id** (*int*) – The SpiNNaker link to which the motor is connected
- **board_address** (*str or None*) –
- **speed** (*int*) –
- **sample_time** (*int*) –
- **update_time** (*int*) –
- **delay_time** (*int*) –
- **delta_threshold** (*int*) –
- **continue_if_not_different** (*bool*) –
- **label** (*str or None*) –

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)

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

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

Return type iterable(str)

class spynnaker8.external_devices.**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

Parameters

- **n_neurons** (*int*) – Number of neurons
- **fpga_link_id** (*int*) –
- **fpga_id** (*int*) –
- **board_address** (*str or None*) –
- **label** (*str or None*) –

class spynnaker8.external_devices.**PushBotRetinaViewer**(*resolution*, *port=0*, *display_max=33.0*, *frame_time_ms=10*, *decay_time_constant_ms=100*)
Bases: threading.Thread

A viewer for the pushbot's retina. This is a thread that can be launched in parallel with the control code.

Based on matplotlib

Parameters

- **resolution** (PushBotRetinaResolution) –
- **port** (*int*) –
- **display_max** (*float*) – Value of brightest pixel to show
- **frame_time_ms** (*int*) – How regularity to display frames (milliseconds)
- **decay_time_constant_ms** (*int*) – Time constant of pixel decay (milliseconds)

local_host

local_port

run()
How the viewer works when the thread is running.

class spynnaker8.external_devices.**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*, *n_steps_per_timestep*, *drop_late_spikes*, *splitter*)

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(AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type ApplicationVertex

class spynnaker8.external_devices.**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** (*int or None*) – The optional instance key to use
- **uart_id** (*int*) – The ID of the UART when needed

class MODES

Bases: enum.Enum

types of modes supported by this protocol

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*, *time=None*)

generic_motor_total_period_key

instance_key

The key of this instance of the protocol

Return type int

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

Return type *MODES*

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** (RetinaKey) – the new key for the retina
- **retina_payload** (RetinaPayload or None) – the new payload for the set retina key command packet
- **time** (int or float or None) – when to transmit this packet

Returns the command to send

Return type MultiCastCommand

set_retina_transmission_key

turn_off_sensor_reporting(*sensor_id*, *time=None*)

turn_off_sensor_reporting_key

uart_id

Return type int

class spynnaker8.external_devices.**PushBotLaser**
Bases: spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice

The properties of the laser device that may be set.

LASER_ACTIVE_TIME = 1
The active period for the laser (no larger than the total period)

LASER_FREQUENCY = 2
The frequency of the laser

LASER_TOTAL_PERIOD = 0
The total period for the laser

class spynnaker8.external_devices.**PushBotLED**
Bases: spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice

The properties of the LED device that may be set.

LED_BACK_ACTIVE_TIME = 2

LED_FREQUENCY = 3

LED_FRONT_ACTIVE_TIME = 1

LED_TOTAL_PERIOD = 0

class spynnaker8.external_devices.**PushBotMotor**
Bases: spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice

The properties of the motor devices that may be set. The pushbot has two motors, 0 (left) and 1 (right).

MOTOR_0_LEAKY = 1
For motor 0, set a variable speed depending on time since event receive

MOTOR_0_PERMANENT = 0
For motor 0, set a particular speed

MOTOR_1_LEAKY = 3
For motor 1, set a variable speed depending on time since event receive

MOTOR_1_PERMANENT = 2
For motor 0, set a particular speed

class spynnaker8.external_devices.**PushBotSpeaker**
Bases: spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_output_device.AbstractPushBotOutputDevice

The properties of the speaker device that may be set.

SPEAKER_ACTIVE_TIME = 1

SPEAKER_MELODY = 3

SPEAKER_TONE = 2

SPEAKER_TOTAL_PERIOD = 0

class spynnaker8.external_devices.**PushBotRetinaResolution**
Bases: enum.Enum

Resolutions supported by the pushbot retina device

DOWNSAMPLE_16_X_16 = <RetinaKey.DOWNSAMPLE_16_X_16: 268435456>
Down sampled 64 (8×8) pixels to 1

DOWNSAMPLE_32_X_32 = <RetinaKey.DOWNSAMPLE_32_X_32: 201326592>
Down sampled 16 (4 × 4) pixels to 1

DOWNSAMPLE_64_X_64 = <RetinaKey.DOWNSAMPLE_64_X_64: 134217728>
Down sampled 4 (2 × 2) pixels to 1

NATIVE_128_X_128 = <RetinaKey.NATIVE_128_X_128: 67108864>
The native resolution

class spynnaker8.external_devices.**PushBotLifEthernet**(***kwargs*)
Bases: spynnaker.pyNN.external_devices_models.external_device_lif_control.ExternalDeviceLifControl

Leaky integrate and fire neuron with an exponentially decaying current input

Parameters

- **protocol** (MunichIoEthernetProtocol) – How to talk to the bot.
- **devices** (*iterable* (AbstractMulticastControllableDevice)) – The devices on the bot that we are interested in.
- **pushbot_ip_address** (*str*) – Where is the pushbot?
- **pushbot_port** (*int*) – (defaulted)
- **tau_m** (*float*) – LIF neuron parameter (defaulted)
- **cm** (*float*) – LIF neuron parameter (defaulted)
- **v_rest** (*float*) – LIF neuron parameter (defaulted)
- **v_reset** (*float*) – LIF neuron parameter (defaulted)
- **tau_syn_E** (*float*) – LIF neuron parameter (defaulted)
- **tau_syn_I** (*float*) – LIF neuron parameter (defaulted)
- **tau_refrac** (*float*) – LIF neuron parameter (defaulted)
- **i_offset** (*float*) – LIF neuron parameter (defaulted)
- **v** (*float*) – LIF neuron parameter (defaulted)
- **isyn_exc** (*float*) – LIF neuron parameter (defaulted)
- **isyn_inh** (*float*) – LIF neuron parameter (defaulted)

class spynnaker8.external_devices.**PushBotEthernetLaserDevice**(*laser, protocol, start_active_time=None, start_total_period=None, start_frequency=None, timesteps_between_send=None*)
Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_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** (PushBotLaser) – The PushBotLaser value to control

- **protocol** (MunichIoEthernetProtocol) – 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.

Parameters **command_protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol to use for this 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)

class spynnaker8.external_devices.**PushBotEthernetLEDDevice** (*led*, *protocol*, *start_active_time_front=None*, *start_active_time_back=None*, *start_total_period=None*, *start_frequency=None*, *timesteps_between_send=None*)

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_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** (PushBotLED) – The PushBotLED parameter to control
- **protocol** (MunichIoEthernetProtocol) – 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.

Parameters **command_protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol to use for this 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)

class spynnaker8.external_devices.**PushBotEthernetMotorDevice** (*motor*, *protocol*, *timesteps_between_send=None*)
Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_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** (PushBotMotor) – a PushBotMotor value to indicate the motor to control
- **protocol** (MunichIoEthernetProtocol) – 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.

Parameters **command_protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol to use for this 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)

class spynnaker8.external_devices.**PushBotEthernetSpeakerDevice**(*speaker, protocol, start_active_time=0, start_total_period=0, start_frequency=0, start_melody=None, timesteps_between_send=None*)

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_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 Speaker of a PushBot

Parameters

- **speaker** (PushBotSpeaker) – The PushBotSpeaker value to control
- **protocol** (MunichIoEthernetProtocol) – The protocol instance to get commands from
- **start_active_time** – The "active time" to set at the start
- **start_total_period** – The "total period" to set at the start
- **start_frequency** – The "frequency" to set at the start
- **start_melody** – The "melody" 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.

Parameters **command_protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol to use for this 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)

class spynnaker8.external_devices.**PushBotEthernetRetinaDevice**(*protocol*, *resolution*, *pushbot_ip_address*, *pushbot_port=56000*, *injector_port=None*, *local_host=None*, *local_port=None*, *retina_injector_label='PushBotRetinaInje*

Bases: spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_retina_device.AbstractPushBotRetinaDevice, spynnaker.pyNN.external_devices_models.abstract_ethernet_sensor.AbstractEthernetSensor

Parameters

- **protocol** (MunichIoEthernetProtocol) –
- **resolution** (PushBotRetinaResolution) –
- **pushbot_ip_address** –
- **pushbot_port** –
- **injector_port** –
- **local_host** –
- **local_port** –
- **retina_injector_label** –

get_database_connection()

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

Return type *SpynnakerLiveSpikesConnection*

Return type *PushBotRetinaConnection*

get_injector_label()

Get the label to give to the Spike Injector

Return type str

get_injector_parameters()

Get the parameters of the Spike Injector to use with this device

Return type dict(str,Any)

get_n_neurons()

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

Return type int

get_translator()

Get a translator of multicast commands to Ethernet commands

Return type *AbstractEthernetTranslator*

class spynnaker8.external_devices.**PushBotLifSpinnakerLink**(***kwargs*)

Bases: spynnaker.pyNN.external_devices_models.external_device_lif_control.ExternalDeviceLifControl

Control module for a PushBot connected to a SpiNNaker Link

Parameters

- **protocol** (MunichIoSpiNNakerLinkProtocol) – How to talk to the bot.
- **devices** (*iterable*(AbstractMulticastControllableDevice)) – The devices on the bot that we are interested in.
- **tau_m** (*float*) – LIF neuron parameter (defaulted)
- **cm** (*float*) – LIF neuron parameter (defaulted)
- **v_rest** (*float*) – LIF neuron parameter (defaulted)
- **v_reset** (*float*) – LIF neuron parameter (defaulted)
- **tau_syn_E** (*float*) – LIF neuron parameter (defaulted)
- **tau_syn_I** (*float*) – LIF neuron parameter (defaulted)
- **tau_refrac** (*float*) – LIF neuron parameter (defaulted)
- **i_offset** (*float*) – LIF neuron parameter (defaulted)
- **v** (*float*) – LIF neuron parameter (defaulted)
- **isyn_exc** (*float*) – LIF neuron parameter (defaulted)
- **isyn_inh** (*float*) – LIF neuron parameter (defaulted)

class spynnaker8.external_devices.**PushBotSpiNNakerLinkLaserDevice**(*laser*, *protocol*, *spinnaker_link_id*, *n_neurons=1*, *label=None*, *board_address=None*, *start_active_time=0*, *start_total_period=0*, *start_frequency=0*)

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_laser_device.PushBotEthernetLaserDevice, pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex

The Laser of a PushBot

Parameters

- **laser** (PushBotLaser) – Which laser device to control
- **protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol instance to get commands from
- **spinnaker_link_id** (*int*) – The SpiNNakerLink that the PushBot is connected to
- **n_neurons** (*int*) – The number of neurons in the device
- **label** (*str*) – A label for the device
- **board_address** (*str or None*) – The IP address of the board that the device is connected to
- **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

default_parameters = {'board_address': None, 'label': None, 'n_neurons': 1, 'start_

class spynnaker8.external_devices.**PushBotSpiNNakerLinkLEDDevice**(*led*, *protocol*, *spinnaker_link_id*, *n_neurons=1*, *label=None*, *board_address=None*, *start_active_time_front=None*, *start_active_time_back=None*, *start_total_period=None*, *start_frequency=None*)

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_led_device.PushBotEthernetLEDDevice, pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex

The LED of a PushBot

Parameters

- **led** (PushBotLED) – The LED device to control
- **protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol instance to get commands from
- **spinnaker_link_id** (*int*) – The SpiNNakerLink connected to
- **n_neurons** (*int*) – The number of neurons in the device
- **label** (*str*) – The label of the device
- **board_address** (*str or None*) – The IP address of the board that the device is connected to
- **start_active_time_front** (*int or None*) – The "active time" to set for the front LED at the start
- **start_active_time_back** (*int or None*) – The "active time" to set for the back LED at the start
- **start_total_period** (*int or None*) – The "total period" to set at the start
- **start_frequency** (*int or None*) – The "frequency" to set at the start

default_parameters = {'board_address': None, 'label': None, 'n_neurons': 1, 'start_

class spynnaker8.external_devices.**PushBotSpiNNakerLinkMotorDevice**(*motor*, *protocol*, *spinnaker_link_id*, *n_neurons=1*, *label=None*, *board_address=None*)

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_motor_device.PushBotEthernetMotorDevice, pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex

The motor of a PushBot

Parameters

- **motor** (PushBotMotor) – the motor to control
- **protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol used to control the device
- **spinnaker_link_id** (*int*) – The SpiNNakerLink connected to
- **n_neurons** (*int*) – The number of neurons in the device
- **label** (*str*) – The label of the device
- **board_address** (*str or None*) – The IP address of the board that the device is connected to

default_parameters = {'board_address': None, 'label': None, 'n_neurons': 1}

class spynnaker8.external_devices.**PushBotSpiNNakerLinkSpeakerDevice**(*speaker, protocol, spinnaker_link_id, n_neurons=1, label=None, board_address=None, start_active_time=50, start_total_period=100, start_frequency=None, start_melody=None*)

Bases: spynnaker.pyNN.external_devices_models.push_bot.ethernet.push_bot_speaker_device.PushBotEthernetSpeakerDevice, pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex

The speaker of a PushBot

Parameters

- **speaker** (PushBotSpeaker) – Which speaker device to control
- **protocol** (MunichIoSpiNNakerLinkProtocol) – The protocol instance to get commands from
- **spinnaker_link_id** (*int*) – The SpiNNakerLink connected to
- **n_neurons** (*int*) – The number of neurons in the device
- **label** (*str*) – The label of the device
- **board_address** (*str or None*) – The IP address of the board that the device is connected to
- **start_active_time** – The "active time" to set at the start
- **start_total_period** – The "total period" to set at the start
- **start_frequency** – The "frequency" to set at the start
- **start_melody** – The "melody" to set at the start

default_parameters = {'board_address': None, 'label': None, 'n_neurons': 1, 'start_

class spynnaker8.external_devices.**PushBotSpiNNakerLinkRetinaDevice**(**args, **kwargs*)

Bases: spynnaker.pyNN.external_devices_models.push_bot.abstract_push_bot_retina_device.AbstractPushBotRetinaDevice, `pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex`

default_parameters = {'board_address': None, 'label': None}

routing_info(*routing_info*)

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

class spynnaker8.external_devices.**SpynnakerLiveSpikesConnection**(*receive_labels=None, send_labels=None, local_host=None, local_port=19999, live_packet_gather_label='LiveSpikeRe*

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(str)*) – Labels of population from which live spikes will be received.
- **send_labels** (*iterable(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

class spynnaker8.external_devices.**SpynnakerPoissonControlConnection**(*poisson_labels=None*, *local_host=None*, *local_port=19999*, *control_label_extension='_control'*)

Bases: `spinn_front_end_common.utilities.connections.live_event_connection.LiveEventConnection`

Parameters

- **poisson_labels** (*iterable(str)*) – Labels of Poisson populations to be controlled
- **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)
- **control_label_extension** (*str*) – The extra name added to the label of each Poisson source

add_init_callback(*label*, *init_callback*)

Add a callback to be called to initialise a vertex

Parameters

- **label** (*str*) – The label of the vertex to be notified about. Must be one of the vertices listed in the constructor
- **init_callback** (*callable(str, int, float, float) -> None*) – A function to be called to initialise the vertex. This should take as parameters the label of the vertex, the number of neurons in the population, the run time of the simulation in milliseconds, and the simulation timestep in milliseconds

add_pause_stop_callback(*label*, *pause_stop_callback*)

Add a callback for the pause and stop state of the simulation

Parameters

- **label** (*str*) – the label of the function to be sent
- **pause_stop_callback** (*callable(str, LiveEventConnection) -> None*) – A function to be called when the pause or stop message has been received. This function should take the label of the referenced vertex, and an instance of this class, which can be used to send events.

Return type None

add_poisson_label(*label*)

Parameters **label** (*str*) – The label of the Poisson source population.

add_receive_callback(*label*, *live_event_callback*, *translate_key=False*)

Add a callback for the reception of live events from a vertex

Parameters

- **label** (*str*) – The label of the vertex to be notified about. Must be one of the vertices listed in the constructor
- **live_event_callback** (*callable(str, int, list(int)) -> None*) – A function to be called when events are received. This should take as parameters the

label of the vertex, the simulation timestep when the event occurred, and an array-like of atom IDs.

- **translate_key** (*bool*) – True if the key is to be converted to an atom ID, False if the key should stay a key

add_start_callback (*label*, *start_callback*)
Add a callback for the start of the simulation

Parameters

- **start_callback** (*callable(str, LiveEventConnection) -> None*) – A function to be called when the start message has been received. This function should take the label of the referenced vertex, and an instance of this class, which can be used to send events
- **label** (*str*) – the label of the function to be sent

add_start_resume_callback (*label*, *start_resume_callback*)
Add a callback for the start and resume state of the simulation

Parameters

- **label** (*str*) – the label of the function to be sent
- **start_resume_callback** (*callable(str, LiveEventConnection) -> None*) – A function to be called when the start or resume message has been received. This function should take the label of the referenced vertex, and an instance of this class, which can be used to send events.

Return type None

set_rate (*label*, *neuron_id*, *rate*)
Set the rate of a Poisson neuron within a Poisson source

Parameters

- **label** (*str*) – The label of the Population to set the rates of
- **neuron_id** (*int*) – The neuron ID to set the rate of
- **rate** (*float*) – The rate to set in Hz

set_rates (*label*, *neuron_id_rates*)
Set the rates of multiple Poisson neurons within a Poisson source

Parameters

- **label** (*str*) – The label of the Population to set the rates of
- **neuron_id_rates** (*list(tuple(int, float))*) – A list of tuples of (neuron ID, rate) to be set

spynnaker8.external_devices.**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** (Population) – 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** (*EIEIOPrefix*) – if the prefix type is 32 bit or 16 bit
- **message_type** (*EIEIOType*) – If the message is a EIEIO command message, or an EIEIO data message with 16 bit or 32 bit keys.
- **payload_as_time_stamps** (*bool*) –
- **right_shift** (*int*) –
- **use_payload_prefix** (*bool*) –
- **notify** (*bool*) –
- **payload_prefix** (*int or None*) –
- **payload_right_shift** (*int*) –
- **number_of_packets_sent_per_time_step** (*int*) –
- **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
- **label** (*str*) – The label of the gatherer vertex
- **partition_ids** (*list(str)*) – The names of the partitions to create edges for

spynnaker8.external_devices.**activate_live_output_to**(*population*, *device*, *partition_id='SPIKE'*)

Activate the output of spikes from a population to an external device. Note that all spikes will be sent to the device.

Parameters

- **population** (Population) – The pyNN population object from which spikes will be sent.
- **device** (Population *or ApplicationVertex*) – The pyNN population or external device to which the spikes will be sent.
- **partition_id** (*str*) – The partition ID to activate live output to.

spynnaker8.external_devices.**SpikeInjector**(*notify=True*, *database_notify_host=None*, *database_notify_port_num=None*, *database_ack_port_num=None*)

Supports creating a spike injector that can be added to the application graph.

Parameters

- **notify** (*bool*) – Whether to register for notifications
- **database_notify_host** (*str or None*) – the hostname for the device which is listening to the database notification.
- **database_ack_port_num** (*int or None*) – 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 or None*) – The port number to which a external device will receive the database is ready command

Returns The spike injector model object that can be placed in a pyNN *Population*.

Return type *AbstractPyNNModel*

spynnaker8.external_devices.**register_database_notification_request**(*hostname*, *notify_port*, *ack_port*)

Adds a socket system which is registered with the notification protocol

Parameters

- **hostname** (*str*) – hostname to connect to

- **notify_port** (*int*) – port num for the notify command
- **ack_port** (*int*) – port num for the acknowledge command

spynnaker8.external_devices.**run_forever**(*sync_time=0*)
Supports running forever in PyNN 0.8/0.9 format

Parameters sync_time – The time in milliseconds after which to pause before the host must continue the simulation

Returns when the application has started running on the SpiNNaker platform

spynnaker8.external_devices.**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** (Population) – 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 an external device will receive the database is ready command
- **notify** (*bool*) – adds to the notification protocol if set.
- **reserve_reverse_ip_tag** (*bool*) – True if a reverse IP tag is to be used, False if SDP is to be used (default)

2.1.1.2 spynnaker8.extra_models package

Module contents

spynnaker8.extra_models.**IFCurDelta**
alias of `spynnaker.pyNN.models.neuron.builds.if_curr_delta.IFCurrDelta`

class spynnaker8.extra_models.**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

Parameters

- **tau_m** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_m
- **cm** (*float, iterable(float), RandomDistribution or (mapping) function*) – C_m
- **v_rest** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{rest}
- **v_reset** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{reset}
- **v_thresh** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{thresh}
- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_e
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_i
- **tau_refrac** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{refrac}
- **i_offset** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_{offset}
- **tau_ca2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $\tau_{\mathrm{Ca}^{+2}}$
- **i_ca2** (*float, iterable(float), RandomDistribution or (mapping) function*) – $I_{\mathrm{Ca}^{+2}}$
- **i_alpha** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_α
- **v** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{init}
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_e
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_i

class spynnaker8.extra_models.**IFCondExpStoc**(***kwargs*)

Bases: spynnaker.pyNN.models.neuron.abstract_pynn_neuron_model_standard.AbstractPyNNNeuronModelStandard

Leaky integrate and fire neuron with a stochastic threshold.

Habenschuss S, Jonke Z, Maass W. Stochastic computations in cortical microcircuit models. *PLoS Computational Biology.* 2013;9(11):e1003311. doi:10.1371/journal.pcbi.1003311

Parameters

- **tau_m** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_m
- **cm** (*float, iterable(float), RandomDistribution or (mapping) function*) – C_m
- **v_rest** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{rest}

- **v_reset** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{reset}
- **v_thresh** (*float, iterable(float), RandomDistribution or (mapping) function*) – V_{thresh}
- **tau_syn_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_e
- **tau_syn_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ^{syn}_i
- **tau_refrac** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{refrac}
- **i_offset** (*float, iterable(float), RandomDistribution or (mapping) function*) – I_{offset}
- **e_rev_E** (*float, iterable(float), RandomDistribution or (mapping) function*) – E^{rev}_e
- **e_rev_I** (*float, iterable(float), RandomDistribution or (mapping) function*) – E^{rev}_i
- **du_th** (*float, iterable(float), RandomDistribution or (mapping) function*) – du_{thresh}
- **tau_th** (*float, iterable(float), RandomDistribution or (mapping) function*) – τ_{thresh}
- **v** (*Float, float, iterable(float), RandomDistribution or (mapping) function*) – V_{init}
- **isyn_exc** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_e
- **isyn_inh** (*float, iterable(float), RandomDistribution or (mapping) function*) – I^{syn}_i

spynnaker8.extra_models.**Izhikevich_cond**
alias of `spynnaker.pyNN.models.neuron.builds.izk_cond_exp_base.IzkCondExpBase`

spynnaker8.extra_models.**IF_curr_dual_exp**
alias of `spynnaker.pyNN.models.neuron.builds.if_curr_dual_exp_base.IFCurrDualExpBase`

spynnaker8.extra_models.**IF_curr_exp_sEMD**
alias of `spynnaker.pyNN.models.neuron.builds.if_curr_exp_semd_base.IFCurrExpSEMDBase`

class spynnaker8.extra_models.**WeightDependenceAdditiveTriplet**(*w_min=0.0, w_max=1.0, A3_plus=0.01, A3_minus=0.01*)

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`

An triplet-based additive weight dependence STDP rule.

Parameters

- **w_min** (*float*) – w^{min}

- **w_max** (*float*) – w^{max}
- **A3_plus** (*float*) – A_3^+
- **A3_minus** (*float*) – A_3^-

A3_minus
A_3^-

Return type float

A3_plus
A_3^+

Return type float

default_parameters = {'A3_minus': 0.01, 'A3_plus': 0.01, 'w_max': 1.0, 'w_min': 0.0

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

Parameters

- **n_synapse_types** (*int*) –
- **n_weight_terms** (*int*) –

Return type int

is_same_as(*weight_dependence*)
Determine if this weight dependence is the same as another

Parameters **weight_dependence** (AbstractWeightDependence) –

Return type bool

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

w_max
w^{max}

Return type float

w_min
w^{min}

Return type float

weight_maximum
The maximum weight that will ever be set in a synapse as a result of this rule

Return type float

write_parameters(*spec*, *machine_time_step*, *weight_scales*, *n_weight_terms*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –

- **machine_time_step** (*int*) – (unused?)
- **weight_scales** (*iterable(float)*) –
- **n_weight_terms** (*int*) –

spynnaker8.extra_models.**PfisterSpikeTriplet**
alias of spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_pfister_spike_triplet.TimingDependencePfisterSpikeTriplet

spynnaker8.extra_models.**SpikeNearestPairRule**
alias of spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spike_nearest_pair.TimingDependenceSpikeNearestPair

spynnaker8.extra_models.**RecurrentRule**
alias of spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_recurrent.TimingDependenceRecurrent

spynnaker8.extra_models.**Vogels2011Rule**
alias of spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_vogels_2011.TimingDependenceVogels2011

class spynnaker8.extra_models.**SpikeSourcePoissonVariable**(*rates*, *starts*, *durations=None*)
Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex(*n_neurons*, *label*, *constraints*, *seed*, *splitter*)
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(AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type ApplicationVertex

default_population_parameters = {'seed': None, 'splitter': 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

2.1.1.3 spynnaker8.models package

Subpackages

spynnaker8.models.connectors package

Module contents

Connectors are objects that describe how neurons in *Population*s are connected to each other.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors* instead.

class spynnaker8.models.connectors.**AllToAllConnector**(*allow_self_connections=True*, *safe=True*, *verbose=None*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.all_to_all_connector.AllToAllConnector

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

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.AllToAllConnector* instead.

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.
- **safe** (*bool*) – if True, check that weights and delays have valid values. If False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

class spynnaker8.models.connectors.**ArrayConnector**(*array*, *safe=True*, *callback=None*, *verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.array_connector.ArrayConnector

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

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.ArrayConnector* instead.

Parameters

- **array** (*ndarray (2, uint8)*) – an array of integers
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

class spynnaker8.models.connectors.**CSAConnector**(*cset*, *safe=True*, *callback=None*, *verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.csa_connector.CSAConnector

A CSA (*Connection Set Algebra*, Djurfeldt 2012) connector.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.CSAConnector* instead.

Parameters

- **cset** (*csa.connset.CSet*) – a connection set description
- **safe** (*bool*) – if True, check that weights and delays have valid values. If False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

class spynnaker8.models.connectors.**DistanceDependentProbabilityConnector**(*d_expression*, *allow_self_connections=True*, *safe=True*, *verbose=False*, *n_connections=None*, *rng=None*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.distance_dependent_probability_connector.DistanceDependentProbabilityConnector

Make connections using a distribution which varies with distance.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.DistanceDependentProbabilityConnector* instead.

Parameters

- **d_expression** (*str*) – 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** (*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.
- **safe** (*bool*) – if True, check that weights and delays have valid values. If False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **n_connections** (*int*) – The number of efferent synaptic connections per neuron.
- **rng** (*NumpyRNG*) – random number generator

- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

class spynnaker8.models.connectors.**FixedNumberPostConnector**(*n, allow_self_connections=True, safe=True, verbose=False, with_replacement=False, rng=None, callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.fixed_number_post_connector.FixedNumberPostConnector

PyNN connector that puts a fixed number of connections on each of the post neurons.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.FixedNumberPostConnector* instead.

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.
- **safe** (*bool*) – Whether to check that weights and delays have valid values; if False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **with_replacement** (*bool*) – if False, once a connection is made, it can't be made again; if True, multiple connections between the same pair of neurons are allowed
- **rng** (*NumpyRNG or None*) – random number generator
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

class spynnaker8.models.connectors.**FixedNumberPreConnector**(*n, allow_self_connections=True, safe=True, verbose=False, with_replacement=False, rng=None, callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.fixed_number_pre_connector.FixedNumberPreConnector

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

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.FixedNumberPreConnector* instead.

Parameters

- **n** (*int*) – number of random pre-synaptic neurons connected to post-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.
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **with_replacement** (*bool*) – if False, once a connection is made, it can't be made again; if True, multiple connections between the same pair of neurons are allowed
- **rng** (*NumpyRNG or None*) – random number generator
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

class spynnaker8.models.connectors.**FixedProbabilityConnector**(*p_connect*, *allow_self_connections=True*, *safe=True*, *verbose=False*, *rng=None*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.fixed_probability_connector.FixedProbabilityConnector

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

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.FixedProbabilityConnector* instead.

Parameters

- **p_connect** (*float*) – a number 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.
- **safe** (*bool*) – if True, check that weights and delays have valid values. If False, this check is skipped.
- **space** (*Space*) – a Space object, needed if you wish to specify distance-dependent weights or delays - not implemented
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **rng** (*NumpyRNG or None*) – random number generator
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

class spynnaker8.models.connectors.**FromFileConnector**(*file*, *distributed=False*, *safe=True*, *callback=None*, *verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.from_file_connector.FromFileConnector

Make connections according to a list read from a file.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.FromFileConnector* instead.

Parameters

- **file** (*str or FileIO*) – Either an open file object or the filename of a file containing a list of connections, in the format required by *FromListConnector*. Column headers, if included in the file, must be specified using a list or tuple, e.g.:

```
# columns = ["i", "j", "weight", "delay", "U", "tau_rec"]
```

  Note that the header requires # at the beginning of the line.

- **distributed** (*bool*) – Basic pyNN says:

  if this is True, then each node will read connections from a file called *filename.x*, where *x* is the MPI rank. This speeds up loading connections for distributed simulations.

  Note: Always leave this as False with sPyNNaker, which is not MPI-based.

- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

  Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

class spynnaker8.models.connectors.**FromListConnector**(*conn_list*, *safe=True*, *verbose=False*, *column_names=None*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.from_list_connector.FromListConnector

Make connections according to a list.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.FromListConnector* instead.

Parameters

- **conn_list** (*list(tuple(int,int,..)) or ndarray*) – a list of tuples, one tuple for each connection. Each tuple should contain: *(pre_idx, post_idx, p1, p2, …, pn)* where *pre_idx* is the index (i.e. order in the Population, not the ID) of the presynaptic neuron, *post_idx* is the index of the postsynaptic neuron, and *p1*, *p2*, etc. are the synaptic parameters (e.g., weight, delay, plasticity parameters).

- **safe** (*bool*) – if True, check that weights and delays have valid values. If False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **column_names** (*tuple(str) or list(str) or None*) – the names of the parameters *p1*, *p2*, etc. If not provided, it is assumed the parameters are *weight, delay* (for backwards compatibility).
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

class spynnaker8.models.connectors.**IndexBasedProbabilityConnector**(*index_expression, allow_self_connections=True, rng=None, safe=True, callback=None, verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.index_based_probability_connector.IndexBasedProbabilityConnector

Create an index-based probability connector. The *index_expression* must depend on the indices *i*, *j* of the populations.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.IndexBasedProbabilityConnector* instead.

Parameters

- **index_expression** (*str*) – A function of the indices of the populations, written as a Python expression; the indices will be given as variables `i` and `j` when the expression is evaluated.
- **allow_self_connections** (*bool*) – allow a neuron to connect to itself
- **rng** (*NumpyRNG or None*) – random number generator
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

class spynnaker8.models.connectors.**MultapseConnector**(*n, allow_self_connections=True, with_replacement=True, safe=True, verbose=False, rng=None*)

Bases: `spynnaker.pyNN.models.neural_projections.connectors.multapse_connector.MultapseConnector`

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* (by default). Uniform selection probability is assumed.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.MultapseConnector* instead.

Parameters

- **n** (*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.
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **rng** (*NumpyRNG or None*) – random number generator

class spynnaker8.models.connectors.**OneToOneConnector**(*safe=True*, *callback=None*)

Bases: `spynnaker.pyNN.models.neural_projections.connectors.one_to_one_connector.OneToOneConnector`

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

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.OneToOneConnector* instead.

Parameters

- **safe** (*bool*) – if True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – a function that will be called with the fractional progress of the connection routine. An example would be *progress_bar.set_level*.

Note: Not supported by sPyNNaker.

class spynnaker8.models.connectors.**SmallWorldConnector**(*degree*, *rewiring*, *allow_self_connections=True*, *n_connections=None*, *rng=None*, *safe=True*, *callback=None*, *verbose=False*)

Bases: `spynnaker.pyNN.models.neural_projections.connectors.small_world_connector.SmallWorldConnector`

Create a connector that uses connection statistics based on the Small World network connectivity model. Note that this is typically used from a population to itself.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.SmallWorldConnector* instead.

Parameters

- **degree** (*float*) – the region length where nodes will be connected locally
- **rewiring** (*float*) – the probability of rewiring each edge
- **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.
- **n_connections** (*int or None*) – if specified, the number of efferent synaptic connections per neuron
- **rng** (*NumpyRNG or None*) – random number generator
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – For PyNN compatibility only.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

class spynnaker8.models.connectors.**KernelConnector**(*shape_pre*, *shape_post*, *shape_kernel*, *weight_kernel=None*, *delay_kernel=None*, *shape_common=None*, *pre_sample_steps_in_post=None*, *pre_start_coords_in_post=None*, *post_sample_steps_in_pre=None*, *post_start_coords_in_pre=None*, *safe=True*, *space=None*, *verbose=False*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.kernel_connector.KernelConnector

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.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neural_projections.connectors.KernelConnector* instead.

Parameters

- **shape_pre** (*tuple(int,int)*) – 2D shape of the pre population (rows/height, cols/width, usually the input image shape)
- **shape_post** (*tuple(int,int)*) – 2D shape of the post population (rows/height, cols/width)
- **shape_kernel** (*tuple(int,int)*) – 2D shape of the kernel (rows/height, cols/width)
- **weight_kernel** (*ndarray or NumpyRNG or int or float or list(int) or list(float) or None*) – (optional) 2D matrix of size shape_kernel describing the weights
- **delay_kernel** (*ndarray or NumpyRNG or int or float or list(int) or list(float) or None*) – (optional) 2D matrix of size shape_kernel describing the delays
- **shape_common** (*tuple(int,int)*) – (optional) 2D shape of common coordinate system (for both pre and post, usually the input image sizes)

- **pre_sample_steps_in_post** (*tuple(int, int)*) – (optional) Sampling steps/jumps for pre pop ⇔ $(\text{step}_x, \text{step}_y)$
- **pre_start_coords_in_post** (*tuple(int, int)*) – (optional) Starting row/col for pre sampling ⇔ $(\text{offset}_x, \text{offset}_y)$
- **post_sample_steps_in_pre** (*tuple(int, int)*) – (optional) Sampling steps/jumps for post pop ⇔ $(\text{step}_x, \text{step}_y)$
- **post_start_coords_in_pre** (*tuple(int, int)*) – (optional) Starting row/col for post sampling ⇔ $(\text{offset}_x, \text{offset}_y)$
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **space** (*Space*) – Currently ignored; for future compatibility.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **callback** (*callable*) – (ignored)

spynnaker8.models.populations package

Module contents

Populations are objects that contain groups of neurons.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.populations* instead.

class spynnaker8.models.populations.**Assembly**(**populations*, ***kwargs*)

Bases: spynnaker.pyNN.models.populations.assembly.Assembly

A group of neurons, may be heterogeneous, in contrast to a Population where all the neurons are of the same type.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.populations.Assembly* instead.

Parameters

- **populations** (Population *or* PopulationView) – the populations or views to form the assembly out of
- **kwargs** – may contain *label* (a string describing the assembly)

class spynnaker8.models.populations.**Population**(*size*, *cellclass*, *cellparams=None*, *structure=None*, *initial_values=None*, *label=None*, *constraints=None*, *additional_parameters=None*)

Bases: spynnaker.pyNN.models.populations.population.Population

PyNN 0.9 population object.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.populations.Population* instead.

Parameters

- **size** (*int*) – The number of neurons in the population
- **cellclass** (*type or* AbstractPyNNModel) – The implementation of the individual neurons.

- **cellparams** (*dict(str,object) or None*) – Parameters to pass to `cellclass` if it is a class to instantiate. Must be `None` if `cellclass` is an instantiated object.
- **structure** (*BaseStructure*) –
- **initial_values** (*dict(str,float)*) – Initial values of state variables
- **label** (*str*) – A label for the population
- **constraints** (*list(AbstractConstraint)*) – Any constraints on how the population is deployed to SpiNNaker.
- **additional_parameters** (*dict(str, ..)*) – Additional parameters to pass to the vertex creation function.

class spynnaker8.models.populations.**PopulationView**(*parent*, *selector*, *label=None*)

Bases: `spynnaker.pyNN.models.populations.population_view.PopulationView`

A view of a subset of neurons within a *Population*.

In most ways, Populations and PopulationViews have the same behaviour, i.e., they can be recorded, connected with Projections, etc. It should be noted that any changes to neurons in a PopulationView will be reflected in the parent Population and *vice versa.*

It is possible to have views of views.

Note: Selector to Id is actually handled by `AbstractSized`.

Deprecated since version 6.0: Use *spynnaker.pyNN.models.populations.PopulationView* instead.

Parameters

- **parent** (Population or PopulationView) – the population or view to make the view from
- **selector** (*None or slice or int or list(bool) or list(int) or ndarray(bool) or ndarray(int)*) – a slice or numpy mask array. The mask array should either be a boolean array (ideally) of the same size as the parent, or an integer array containing cell indices, i.e. if *p.size == 5* then:

```
PopulationView(p, array([False, False, True, False, True]))
PopulationView(p, array([2, 4]))
PopulationView(p, slice(2, 5, 2))
```

  will all create the same view.
- **label** (*str*) – A label for the view

spynnaker8.models.synapse_dynamics package

Subpackages

spynnaker8.models.synapse_dynamics.timing_dependence package

Module contents

> **Warning:** Using classes via this module is deprecated. Please use *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence* instead.

class spynnaker8.models.synapse_dynamics.timing_dependence.**TimingDependenceSpikePair**(*tau_plus*, *tau_min*, *A_plus=*, *A_minu*

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spike_pair.TimingDependenceSpikePair

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.TimingDependenceSpikePair* instead.

Parameters

- **tau_plus** (*float*) – τ_+
- **tau_minus** (*float*) – τ_-
- **A_plus** (*float*) – A^+
- **A_minus** (*float*) – A^-

class spynnaker8.models.synapse_dynamics.timing_dependence.**TimingDependencePfisterSpikeTri**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_pfister_spike_triplet.TimingDependencePfisterSpikeTriplet

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.TimingDependencePfisterSpikeTriplet* instead.

Parameters

- **tau_plus** (*float*) – τ_+
- **tau_minus** (*float*) – τ_-
- **tau_x** (*float*) – τ_x
- **tau_y** (*float*) – τ_y
- **A_plus** (*float*) – A^+
- **A_minus** (*float*) – A^-

class spynnaker8.models.synapse_dynamics.timing_dependence.**TimingDependenceRecurrent**(*accumu 6, ac- cu- mu- la- tor_pote mean_p mean_p dual_fsn A_plus= A_minu*

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_recurrent.TimingDependenceRecurrent

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.TimingDependenceRecurrent* instead.

Parameters

- **accumulator_depression** (*int*) –
- **accumulator_potentiation** (*int*) –
- **mean_pre_window** (*float*) –
- **mean_post_window** (*float*) –
- **dual_fsm** (*bool*) –
- **A_plus** (*float*) – A^+
- **A_minus** (*float*) – A^-

class spynnaker8.models.synapse_dynamics.timing_dependence.**TimingDependenceSpikeNearestPai**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spike_nearest_pair.TimingDependenceSpikeNearestPair

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.TimingDependenceSpikeNearestPair* instead.

Parameters

- **tau_plus** (*float*) – τ_+
- **tau_minus** (*float*) – τ_-
- **A_plus** (*float*) – A^+
- **A_minus** (*float*) – A^-

class spynnaker8.models.synapse_dynamics.timing_dependence.**TimingDependenceVogels2011**(*alpha, tau=2 A_plus A_min*

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_vogels_2011.TimingDependenceVogels2011

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.TimingDependenceVogels2011* instead.

Parameters

- **alpha** (*float*) – α
- **tau** (*float*) – τ
- **A_plus** (*float*) – A^{+}
- **A_minus** (*float*) – A^{-}

spynnaker8.models.synapse_dynamics.weight_dependence package

Module contents

> **Warning:** Using classes via this module is deprecated. Please use *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence* instead.

class spynnaker8.models.synapse_dynamics.weight_dependence.**WeightDependenceAdditive**(*w_min=0*
w_max=1

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_additive.WeightDependenceAdditive

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.WeightDependenceAdditive* instead.

Parameters

- **w_min** (*float*) – $w_{\min}$
- **w_max** (*float*) – $w_{\max}$

class spynnaker8.models.synapse_dynamics.weight_dependence.**WeightDependenceMultiplicative**(*w*

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_multiplicative.WeightDependenceMultiplicative

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.WeightDependenceMultiplicative* instead.

Parameters

- **w_min** (*float*) – $w_{\min}$
- **w_max** (*float*) – $w_{\max}$

class spynnaker8.models.synapse_dynamics.weight_dependence.**WeightDependenceAdditiveTriplet**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_additive_triplet.WeightDependenceAdditiveTriplet

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.WeightDependenceAdditiveTriplet* instead.

Parameters

- **w_min** (*float*) – $w_{\min}$
- **w_max** (*float*) – $w_{\max}$

- **A3_plus** (*float*) – A_3^+
- **A3_minus** (*float*) – A_3^-

Module contents

> **Warning:** Using classes via this module is deprecated. Please use *spynnaker.pyNN.models.neuron.synapse_dynamics* instead.

class spynnaker8.models.synapse_dynamics.**SynapseDynamicsStatic**(*weight=<sphinx.ext.autodoc.importer._M object>, delay=None*)

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_static.SynapseDynamicsStatic

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.synapse_dynamics.SynapseDynamicsStatic* instead.

Parameters

- **weight** (*float*) –
- **delay** (*float or None*) –

class spynnaker8.models.synapse_dynamics.**SynapseDynamicsSTDP**(*timing_dependence, weight_dependence, voltage_dependence=None, dendritic_delay_fraction=1.0, weight=<sphinx.ext.autodoc.importer._Moc object>, delay=None, backprop_delay=True*)

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_stdp.SynapseDynamicsSTDP

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.synapse_dynamics.SynapseDynamicsSTDP* instead.

Parameters

- **timing_dependence** (AbstractTimingDependence) –
- **weight_dependence** (AbstractWeightDependence) –
- **voltage_dependence** (*None*) – Unsupported
- **dendritic_delay_fraction** (*float*) –
- **weight** (*float*) –
- **delay** (*float or None*) –
- **backprop_delay** (*bool*) –

class spynnaker8.models.synapse_dynamics.**SynapseDynamicsStructuralStatic**(*partner_selection*, *formation*, *elimination*, *f_rew=10000.0*, *initial_weight=0.0*, *initial_delay=1.0*, *s_max=32*, *seed=None*, *weight=<sphinx.ext.autod... object>*, *delay=None*)

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_static.SynapseDynamicsStructuralStatic

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.synapse_dynamics.SynapseDynamicsStructuralStatic* instead.

Parameters

- **partner_selection** (AbstractPartnerSelection) – The partner selection rule
- **formation** (AbstractFormation) – The formation rule
- **elimination** (AbstractElimination) – The elimination rule
- **f_rew** (*int*) – How many rewiring attempts will be done per second.
- **initial_weight** (*float*) – Weight assigned to a newly formed connection
- **initial_delay** (*float or tuple(float, float)*) – Delay assigned to a newly formed connection; a single value means a fixed delay value, or a tuple of two values means the delay will be chosen at random from a uniform distribution between the given values
- **s_max** (*int*) – Maximum fan-in per target layer neuron
- **seed** (*int*) – seed the random number generators
- **weight** (*float*) – The weight of connections formed by the connector
- **delay** (*float or None*) – The delay of connections formed by the connector

class spynnaker8.models.synapse_dynamics.**SynapseDynamicsStructuralSTDP**(*partner_selection, formation, elimination, timing_dependence=None, weight_dependence=None, voltage_dependence=None, dendritic_delay_fraction=1.0, f_rew=10000.0, initial_weight=0.0, initial_delay=1.0, s_max=32, seed=None, weight=<sphinx.ext.autodoc. object>, delay=None, backprop_delay=True*)

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_stdp.SynapseDynamicsStructuralSTDP

Deprecated since version 6.0: Use *spynnaker.pyNN.models.neuron.synapse_dynamics.SynapseDynamicsStructuralSTDP* instead.

Parameters

- **partner_selection** (AbstractPartnerSelection) – The partner selection rule
- **formation** (AbstractFormation) – The formation rule
- **elimination** (AbstractElimination) – The elimination rule
- **timing_dependence** (AbstractTimingDependence) – The STDP timing dependence rule
- **weight_dependence** (AbstractWeightDependence) – The STDP weight dependence rule
- **voltage_dependence** (*None*) – The STDP voltage dependence (unsupported)
- **dendritic_delay_fraction** (*float*) – The STDP dendritic delay fraction
- **f_rew** (*int*) – How many rewiring attempts will be done per second.
- **initial_weight** (*float*) – Weight assigned to a newly formed connection
- **initial_delay** (*float or tuple(float, float)*) – Delay assigned to a newly formed connection; a single value means a fixed delay value, or a tuple of two values means the delay will be chosen at random from a uniform distribution between the given values

- **s_max** (*int*) – Maximum fan-in per target layer neuron
- **seed** (*int*) – seed the random number generators
- **weight** (*float*) – The weight of connections formed by the connector
- **delay** (*float or None*) – The delay of connections formed by the connector

Module contents

class spynnaker8.models.**Projection**(*pre_synaptic_population*, *post_synaptic_population*, *connector*, *synapse_type=None*, *source=None*, *receptor_type=None*, *space=None*, *label=None*)

Bases: *spynnaker.pyNN.models.projection.Projection*

sPyNNaker 8 projection class

Deprecated since version 6.0: Use *spynnaker.pyNN.models.projection.Projection* instead.

Parameters

- **pre_synaptic_population** (PopulationBase) –
- **post_synaptic_population** (PopulationBase) –
- **connector** (AbstractConnector) –
- **synapse_type** (AbstractStaticSynapseDynamics) –
- **source** (*None*) – Unsupported; must be None
- **receptor_type** (*str*) –
- **space** (*Space*) –
- **label** (*str*) –

class spynnaker8.models.**Recorder**(*population*)

Bases: *spynnaker.pyNN.models.recorder.Recorder*

Deprecated since version 6.0: Use *spynnaker.pyNN.models.recorder.Recorder* instead.

Parameters **population** (Population) – the population to record for

2.1.1.4 spynnaker8.utilities package

Submodules

spynnaker8.utilities.neo_convertor module

spynnaker8.utilities.neo_convertor.**convert_analog_signal**(*signal_array*, *time_unit*)

Converts part of a NEO object into told spynnaker7 format

Deprecated since version 6.0: Use *spynnaker.pyNN.utilities.neo_convertor* instead.

Parameters

- **signal_array** (*AnalogSignal*) – Extended Quantities object
- **time_unit** (*quantities.unitquantity.UnitTime*) – Data time unit for time index

Return type ndarray

spynnaker8.utilities.neo_convertor.**convert_data**(*data*, *name*, *run=0*)
Converts the data into a numpy array in the format ID, time, value

Deprecated since version 6.0: Use *spynnaker.pyNN.utilities.neo_convertor* instead.

Parameters

- **data** (*Block*) – Data as returned by a getData() call
- **name** (*str*) – Name of the data to be extracted. Same values as used in getData()
- **run** (*int*) – Zero based index of the run to extract data for

Return type ndarray

spynnaker8.utilities.neo_convertor.**convert_data_list**(*data*, *name*, *runs=None*)
Converts the data into a list of numpy arrays in the format ID, time, value

Deprecated since version 6.0: Use *spynnaker.pyNN.utilities.neo_convertor* instead.

Parameters

- **data** (*Block*) – Data as returned by a getData() call
- **name** (*str*) – Name of the data to be extracted. Same values as used in getData()
- **runs** (*list (int) or None*) – List of Zero based index of the run to extract data for. Or None to extract all runs

Return type list(ndarray)

spynnaker8.utilities.neo_convertor.**convert_gsyn**(*gsyn_exc*, *gsyn_inh*)
Converts two neo objects into the spynnaker7 format

Deprecated since version 6.0: Use *spynnaker.pyNN.utilities.neo_convertor* instead.

Parameters

- **gsyn_exc** (*Block*) – neo with gsyn_exc data
- **gsyn_inh** (*Block*) – neo with gsyn_exc data

Return type ndarray

spynnaker8.utilities.neo_convertor.**convert_gsyn_exc_list**(*data*, *runs=None*)
Converts the gsyn_exc into a list numpy array one per segment (all runs) in the format ID, time, value

Deprecated since version 6.0: Use *spynnaker.pyNN.utilities.neo_convertor* instead.

Parameters

- **data** (*Block*) – The data to convert; it must have Gsyn_exc data in it
- **runs** (*list (int) or None*) – List of Zero based index of the run to extract data for. Or None to extract all runs

Return type list(ndarray)

spynnaker8.utilities.neo_convertor.**convert_gsyn_inh_list**(*data*, *runs=None*)
Converts the gsyn_inh into a list numpy array one per segment (all runs) in the format ID, time, value

Deprecated since version 6.0: Use *spynnaker.pyNN.utilities.neo_convertor* instead.

Parameters

- **data** (*Block*) – The data to convert; it must have Gsyn_inh data in it
- **runs** (*list (int) or None*) – List of Zero based index of the run to extract data for. Or None to extract all runs

Return type list(ndarray)

`spynnaker8.utilities.neo_convertor.`**`convert_spikes`**(*neo*, *run=0*)

Extracts the spikes for run one from a Neo Object

Deprecated since version 6.0: Use `spynnaker.pyNN.utilities.neo_convertor` instead.

Parameters

- **`neo`** (`Block`) – neo Object including Spike Data
- **`run`** (`int`) – Zero based index of the run to extract data for

Return type ndarray

`spynnaker8.utilities.neo_convertor.`**`convert_spiketrains`**(*spiketrains*)

Converts a list of spiketrains into spynnaker7 format

Deprecated since version 6.0: Use `spynnaker.pyNN.utilities.neo_convertor` instead.

Parameters **`spiketrains`** (`list(SpikeTrain)`) – List of SpikeTrains

Return type ndarray

`spynnaker8.utilities.neo_convertor.`**`convert_v_list`**(*data*, *runs=None*)

Converts the voltage into a list numpy array one per segment (all runs) in the format ID, time, value

Deprecated since version 6.0: Use `spynnaker.pyNN.utilities.neo_convertor` instead.

Parameters

- **`data`** (`Block`) – The data to convert; it must have V data in it
- **`runs`** (`list(int) or None`) – List of Zero based index of the run to extract data for. Or None to extract all runs

Return type list(ndarray)

`spynnaker8.utilities.neo_convertor.`**`count_spikes`**(*neo*)

Help function to count the number of spikes in a list of spiketrains

Deprecated since version 6.0: Use `spynnaker.pyNN.utilities.neo_convertor` instead.

Parameters **`neo`** (`Block`) – Neo Object which has spikes in it

Returns The number of spikes in the first segment

`spynnaker8.utilities.neo_convertor.`**`count_spiketrains`**(*spiketrains*)

Help function to count the number of spikes in a list of spiketrains

Deprecated since version 6.0: Use `spynnaker.pyNN.utilities.neo_convertor` instead.

Parameters **`spiketrains`** (`list(SpikeTrain)`) – List of SpikeTrains

Returns Total number of spikes in all the spiketrains

Return type int

Module contents

2.1.2 Submodules

2.1.3 spynnaker8.spynnaker_plotting module

class spynnaker8.spynnaker_plotting.**SpynnakerPanel**(**data*, ***options*)
Bases: `spynnaker.spynnaker_plotting.SpynnakerPanel`

Represents a single panel in a multi-panel figure.

Deprecated since version 6.0: Use `spynnaker.spynnaker_plotting` instead.

spynnaker8.spynnaker_plotting.**heat_plot_neo**(*ax*, *signal_array*, *label=''*, ***options*)
Plots neurons, times and values into a heatmap

Deprecated since version 6.0: Use `spynnaker.spynnaker_plotting` instead.

Parameters

- **ax** (*Axes*) – An Axes in a matplotlib figure
- **signal_array** (*AnalogSignal*) – Neo Signal array Object
- **label** (*str*) – Label for the graph
- **options** – plotting options

spynnaker8.spynnaker_plotting.**heat_plot_numpy**(*ax*, *data*, *label=''*, ***options*)
Plots neurons, times and values into a heatmap

Deprecated since version 6.0: Use `spynnaker.spynnaker_plotting` instead.

Parameters

- **ax** (*Axes*) – An Axes in a matplotlib figure
- **data** (*ndarray*) – nparray of values in spynnaker7 format
- **label** (*str*) – Label for the graph
- **options** – plotting options

spynnaker8.spynnaker_plotting.**plot_segment**(*axes*, *segment*, *label=''*, ***options*)
Plots a segment into a plot of spikes or a heatmap

Deprecated since version 6.0: Use `spynnaker.spynnaker_plotting` instead.

Parameters

- **axes** (*Axes*) – An Axes in a matplotlib figure
- **segment** (*Segment*) – Data for one run to plot
- **label** (*str*) – Label for the graph
- **options** – plotting options

spynnaker8.spynnaker_plotting.**plot_spikes_numpy**(*ax*, *spikes*, *label=''*, ***options*)
Plot all spikes

Deprecated since version 6.0: Use `spynnaker.spynnaker_plotting` instead.

Parameters

- **ax** (*Axes*) – An Axes in a matplotlib figure

- **spikes** (*ndarray*) – spynakker7 format nparray of spikes
- **label** (*str*) – Label for the graph
- **options** – plotting options

spynnaker8.spynnaker_plotting.**plot_spiketrains**(*ax*, *spiketrains*, *label=''*, ***options*)
Plot all spike trains in a Segment in a raster plot.

Deprecated since version 6.0: Use `spynnaker.spynnaker_plotting` instead.

Parameters

- **ax** (*Axes*) – An Axes in a matplotlib figure
- **spiketrains** (*list (SpikeTrain)*) – List of spiketimes
- **label** (*str*) – Label for the graph
- **options** – plotting options

2.1.4 Module contents

The *spynnaker.pyNN* package contains the front end specifications and implementation for the PyNN High-level API (http://neuralensemble.org/trac/PyNN).

This package contains the profile of that code for PyNN 0.9

spynnaker8.**Cuboid**
Used by autodoc_mock_imports.

spynnaker8.**distance**(*src*, *tgt*, *mask=None*, *scale_factor=1.0*, *offset=0.0*, *periodic_boundaries=None*)
Return the Euclidian distance between two cells.

Parameters

- **src** –
- **tgt** –
- **mask** (*ndarray*) – allows only certain dimensions to be considered, e.g.: * to ignore the z-dimension, use `mask=array([0,1])` * to ignore y, `mask=array([0,2])` * to just consider z-distance, `mask=array([2])`
- **scale_factor** (*float*) – allows for different units in the pre- and post-position (the post-synaptic position is multiplied by this quantity).
- **offset** (*float*) –
- **periodic_boundaries** –

spynnaker8.**Grid2D**
Used by autodoc_mock_imports.

spynnaker8.**Grid3D**
Used by autodoc_mock_imports.

spynnaker8.**Line**
Used by autodoc_mock_imports.

spynnaker8.**NumpyRNG**
Used by autodoc_mock_imports.

class spynnaker8.**RandomDistribution**(**args, **kwargs*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Class which defines a next(n) method which returns an array of `n` random numbers from a given distribution.

Parameters

- **distribution** (*str*) – the name of a random number distribution.
- **parameters_pos** (*tuple or None*) – parameters of the distribution, provided as a tuple. For the correct ordering, see *random.available_distributions*.
- **rng** (*NumpyRNG or GSLRNG or NativeRNG or None*) – the random number generator to use, if a specific one is desired (e.g., to provide a seed).
- **parameters_named** – parameters of the distribution, provided as keyword arguments.

Parameters may be provided either through `parameters_pos` or through `parameters_named`, but not both. All parameters must be provided, there are no default values. Parameter names are, in general, as used in Wikipedia.

Examples:

```
>>> rd = RandomDistribution('uniform', (-70, -50))
>>> rd = RandomDistribution('normal', mu=0.5, sigma=0.1)
>>> rng = NumpyRNG(seed=8658764)
>>> rd = RandomDistribution('gamma', k=2.0, theta=5.0, rng=rng)
```

Table 1: Available distributions

Name	Parameters	Comments
`binomial`	`n`, `p`	
`gamma`	`k`, `theta`	
`exponential`	`beta`	
`lognormal`	`mu`, `sigma`	
`normal`	`mu`, `sigma`	
`normal_clipped`	`mu`, `sigma`, `low`, `high`	Values outside (`low`, `high`) are redrawn
`normal_clipped_to_boundary`	`mu`, `sigma`, `low`, `high`	Values below/above `low`/`high` are set to `low`/`high`
`poisson`	`lambda_`	Trailing underscore since `lambda` is a Python keyword
`uniform`	`low`, `high`	
`uniform_int`	`low`, `high`	Only generates integer values
`vonmises`	`mu`, `kappa`	

spynnaker8.**RandomStructure**
Used by autodoc_mock_imports.

spynnaker8.**Space**
Used by autodoc_mock_imports.

spynnaker8.**Sphere**
Used by autodoc_mock_imports.

class spynnaker8.**AllToAllConnector**(*allow_self_connections=True, safe=True, verbose=None, callback=None*)
Bases: `spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine`,

spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_
AbstractConnectorSupportsViewsOnMachine

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.
- **safe** (*bool*) – If `True`, check that weights and delays have valid values. If `False`, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

allow_self_connections

Return type bool

create_synaptic_block (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id

The ID of the connection generator on the machine.

Return type int

gen_connector_params (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Get the parameters of the on machine generation.

Parameters

- **pre_slices** (*list(Slice)*) –

- **post_slices** (*list* (*Slice*)) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum (*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum (*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list (int) or list (float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum (*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

class spynnaker8.**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** (*ndarray (2, uint8)*) – An explicit boolean matrix that specifies the connections between the pre- and post-populations (see PyNN documentation). Must be 2D in practice.
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

  Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

get_delay_maximum(*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)

Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_weight_maximum(*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (SynapseInformation) –

Return type float

class spynnaker8.**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.

Note: If you get TypeError in Python 3 see: https://github.com/INCF/csa/issues/10

Parameters

- **cset** (*csa.connset.CSet*) – A description of the connection set between populations
- **safe** (*bool*) – If `True`, check that weights and delays have valid values. If `False`, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

  Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

Raises `ImportError` – if the *csa* library isn't present; it's tricky to install in some environments so we don't force it to be present unless you want to actually use this class.

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (`AbstractSynapseType`) –
- **synapse_info** (`SynapseInformation`) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

get_delay_maximum(*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (`SynapseInformation`) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)

Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (`SynapseInformation`) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (`SynapseInformation`) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_weight_maximum(*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (SynapseInformation) –

Return type float

show_connection_set(*n_pre_neurons*, *n_post_neurons*)

Parameters

- **n_pre_neurons** (*int*) –
- **n_post_neurons** (*int*) –

class spynnaker8.**DistanceDependentProbabilityConnector**(*d_expression*, *allow_self_connections=True*, *safe=True*, *verbose=False*, *n_connections=None*, *rng=None*, *callback=None*)

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

Make connections using a distribution which varies with distance.

Parameters

- **d_expression** (*str*) – 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** (*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.
- **safe** (*bool*) – if `True`, check that weights and delays have valid values. If `False`, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **n_connections** (*int or None*) – The number of efferent synaptic connections per neuron.
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or `None` to make one when needed.
- **callback** (*callable*) –

allow_self_connections

Return type bool

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

d_expression
The distance expression.

Return type str

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_weight_maximum(*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (SynapseInformation) –

Return type float

set_projection_information(*machine_time_step*, *synapse_info*)

sets a connectors projection info :param int machine_time_step: machine time step :param SynapseInformation synapse_info: the synapse info

class spynnaker8.**FixedNumberPostConnector**(*n*, *allow_self_connections=True*, *safe=True*, *verbose=False*, *with_replacement=False*, *rng=None*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine, spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_on_machine.AbstractConnectorSupportsViewsOnMachine

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.
- **safe** (*bool*) – Whether to check that weights and delays have valid values; if False, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **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.
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or None to make one when needed.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

allow_self_connections

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id
The ID of the connection generator on the machine.

Return type int

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

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum (*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_weight_maximum(*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (SynapseInformation) –

Return type float

set_projection_information(*machine_time_step*, *synapse_info*)

sets a connectors projection info :param int machine_time_step: machine time step :param SynapseInformation synapse_info: the synapse info

class spynnaker8.**FixedNumberPreConnector**(*n*, *allow_self_connections=True*, *safe=True*, *verbose=False*, *with_replacement=False*, *rng=None*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine, spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_o AbstractConnectorSupportsViewsOnMachine

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.
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **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.
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or None to make one when needed
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

allow_self_connections

create_synaptic_block (*pre_slices, post_slices, pre_vertex_slice, post_vertex_slice, synapse_type, synapse_info*)
Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id
The ID of the connection generator on the machine.

Return type int

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

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –

- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

set_projection_information(*machine_time_step*, *synapse_info*)
sets a connectors projection info :param int machine_time_step: machine time step :param SynapseInformation synapse_info: the synapse info

class spynnaker8.**FixedProbabilityConnector**(*p_connect*, *allow_self_connections=True*, *safe=True*, *verbose=False*, *rng=None*, *callback=None*)

Bases: `spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine`, `spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_on_machine.AbstractConnectorSupportsViewsOnMachine`

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

Parameters

- **p_connect** (*float*) – a value 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.
- **safe** (*bool*) – If `True`, check that weights and delays have valid values. If `False`, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or None to make one when needed
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (`AbstractSynapseType`) –
- **synapse_info** (`SynapseInformation`) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id

The ID of the connection generator on the machine.

Return type int

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

Parameters

- **pre_slices** (*list* (*Slice*)) –
- **post_slices** (*list* (*Slice*)) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes
The size of the connector parameters in bytes.

Return type int

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum (*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum (*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list (int) or list (float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum (*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

p_connect

class spynnaker8.**FromFileConnector** (*file*, *distributed=False*, *safe=True*, *callback=None*, *verbose=False*)
Bases: spynnaker.pyNN.models.neural_projections.connectors.from_list_connector.FromListConnector

Make connections according to a list read from a file.

Parameters

- **file** (*str or FileIO*) – Either an open file object or the filename of a file containing a list of connections, in the format required by *FromListConnector*. Column headers, if included in the file, must be specified using a list or tuple, e.g.:

```
# columns = ["i", "j", "weight", "delay", "U", "tau_rec"]
```

  Note that the header requires # at the beginning of the line.

- **distributed** (*bool*) – Basic pyNN says:

  if this is True, then each node will read connections from a file called filename.x, where x is the MPI rank. This speeds up loading connections for distributed simulations.

  Note: Always leave this as False with sPyNNaker, which is not MPI-based.

- **safe** (*bool*) – Whether to check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

  Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

get_reader (*file*)
Get a file reader object using the PyNN methods.

Returns A pynn StandardTextFile or similar

Return type StandardTextFile

class spynnaker8.**FromListConnector** (*conn_list*, *safe=True*, *verbose=False*, *column_names=None*, *callback=None*)
Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector

Make connections according to a list.

Parameters

- **conn_list** (*ndarray or list(tuple(int,int,..))*) – A numpy array or a list of tuples, one tuple for each connection. Each tuple should contain:

```
(pre_idx, post_idx, p1, p2, ..., pn)
```

  where `pre_idx` is the index (i.e. order in the Population, not the ID) of the presynaptic neuron, `post_idx` is the index of the postsynaptic neuron, and `p1`, `p2`, etc. are the synaptic parameters (e.g., weight, delay, plasticity parameters). All tuples/rows must have the same number of items.

- **safe** (*bool*) – if `True`, check that weights and delays have valid values. If `False`, this check is skipped.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **column_names** (*None or tuple(str) or list(str)*) – the names of the parameters `p1`, `p2`, etc. If not provided, it is assumed the parameters are `weight, delay` (for backwards compatibility).
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

Note: Not supported by sPyNNaker.

column_names

The names of the columns in the array after the first two. Of particular interest is whether `weight` and `delay` columns are present.

Return type list(str)

conn_list

The connection list.

Return type ndarray

could_connect (*_synapse_info*, *_pre_slice*, *_post_slice*)

Checks if a pre slice and a post slice could connect.

Typically used to determine if a Machine Edge should be created by checking that at least one of the indexes in the pre slice could over time connect to at least one of the indexes in the post slice.

Note: This method should never return a false negative, but may return a false positives

Parameters

- **_pre_slice** (*Slice*) –
- **_post_slice** (*Slice*) –
- **_synapse_info** (`SynapseInformation`) –

Return type bool

create_synaptic_block (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

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

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum (*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

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

Parameters **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type float

get_extra_parameter_names ()
Getter for the names of the extra parameters.

Return type list(str)

get_extra_parameters ()
Getter for the extra parameters. Excludes `weight` and `delay` columns.

Returns The extra parameters

Return type ndarray

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

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_hi** (*int*) –
- **post_hi** (*int*) –

Return type int

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_weight_maximum(*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (SynapseInformation) –

Return type float

get_weight_mean(*weights*, *synapse_info*)

Get the mean of the weights.

Parameters **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type float

get_weight_variance(*weights*, *synapse_info*)

Get the variance of the weights.

Parameters **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type float

class spynnaker8.**IndexBasedProbabilityConnector**(*index_expression*, *allow_self_connections=True*, *rng=None*, *safe=True*, *callback=None*, *verbose=False*)

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** (*str*) – 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; the indices will be given as variables `i` and `j` when the expression is evaluated.
- **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.
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or `None` to make one when needed.
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If `False`, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

  Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

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.

Return type bool

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (`AbstractSynapseType`) –
- **synapse_info** (`SynapseInformation`) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

get_delay_maximum(*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (`SynapseInformation`) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

index_expression
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.

Return type str

spynnaker8.**FixedTotalNumberConnector**
alias of spynnaker.pyNN.models.neural_projections.connectors.multapse_connector.MultapseConnector

class spynnaker8.**KernelConnector**(*shape_pre*, *shape_post*, *shape_kernel*, *weight_kernel=None*, *delay_kernel=None*, *shape_common=None*, *pre_sample_steps_in_post=None*, *pre_start_coords_in_post=None*, *post_sample_steps_in_pre=None*, *post_start_coords_in_pre=None*, *safe=True*, *space=None*, *verbose=False*, *callback=None*)

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** (*list(int) or tuple(int,int)*) – 2D shape of the pre population (rows/height, cols/width, usually the input image shape)
- **shape_post** (*list(int) or tuple(int,int)*) – 2D shape of the post population (rows/height, cols/width)
- **shape_kernel** (*list(int) or tuple(int,int)*) – 2D shape of the kernel (rows/height, cols/width)
- **weight_kernel** (*ndarray or NumpyRNG or int or float or list(int) or list(float) or None*) – (optional) 2D matrix of size shape_kernel describing the weights
- **delay_kernel** (*ndarray or NumpyRNG or int or float or list(int) or list(float) or None*) – (optional) 2D matrix of size shape_kernel describing the delays
- **shape_common** (*list(int) or tuple(int,int) or None*) – (optional) 2D shape of common coordinate system (for both pre and post, usually the input image sizes)
- **pre_sample_steps_in_post** (*None or list(int) or tuple(int,int)*) – (optional) Sampling steps/jumps for pre pop <=> (stepX, stepY)
- **pre_start_coords_in_post** (*None or list(int) or tuple(int,int)*) – (optional) Starting row/col for pre sampling <=> (offX, offY)
- **post_sample_steps_in_pre** (*None or list(int) or tuple(int,int)*) – (optional) Sampling steps/jumps for post pop <=> (stepX, stepY)
- **post_start_coords_in_pre** (*None or list(int) or tuple(int,int)*) – (optional) Starting row/col for post sampling <=> (offX, offY)
- **safe** (*bool*) – Whether to check that weights and delays have valid values. If `False`, this check is skipped.
- **space** (*Space*) – Currently ignored; for future compatibility.
- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file
- **callback** (*callable*) – (ignored)

create_synaptic_block(*pre_slices, post_slices, pre_vertex_slice, post_vertex_slice, synapse_type, synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id

The ID of the connection generator on the machine.

Return type int

gen_connector_params (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Get the parameters of the on machine generation.

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(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

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –

Return type ndarray(uint32)

gen_delay_params_size_in_bytes (*delays*)

The size of the delay parameters in bytes

Parameters **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type int

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

Parameters delays (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type int

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

Parameters weights (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type int

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

Parameters weights (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –

Return type int

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

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –

Return type ndarray(uint32)

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

Parameters synapse_info (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters synapse_info (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice, synapse_info, 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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –

- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum (*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_weight_maximum (*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (SynapseInformation) –

Return type float

class spynnaker8.**OneToOneConnector** (*safe=True, callback=None, verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_generate_connector_on_machine.AbstractGenerateConnectorOnMachine, spynnaker.pyNN.models.neural_projections.connectors.abstract_connector_supports_views_o AbstractConnectorSupportsViewsOnMachine

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

Parameters

- **safe** (*bool*) – If True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

  Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

could_connect (*_synapse_info, _pre_slice, _post_slice*)

Checks if a pre slice and a post slice could connect.

Typically used to determine if a Machine Edge should be created by checking that at least one of the indexes in the pre slice could over time connect to at least one of the indexes in the post slice.

Note: This method should never return a false negative, but may return a false positives

Parameters

- **_pre_slice** (*Slice*) –
- **_post_slice** (*Slice*) –
- **_synapse_info** (SynapseInformation) –

Return type bool

create_synaptic_block (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

gen_connector_id

The ID of the connection generator on the machine.

Return type int

gen_connector_params (*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Get the parameters of the on machine generation.

Parameters

- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (AbstractSynapseType) –
- **synapse_info** (SynapseInformation) –

Return type ndarray(uint32)

gen_connector_params_size_in_bytes

The size of the connector parameters in bytes.

Return type int

get_delay_maximum (*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)
Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (SynapseInformation) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

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

Parameters **synapse_info** (SynapseInformation) –

Return type float

use_direct_matrix(*synapse_info*)

Parameters **synapse_info** (SynapseInformation) –

Return type bool

class spynnaker8.**SmallWorldConnector**(*degree*, *rewiring*, *allow_self_connections=True*, *n_connections=None*, *rng=None*, *safe=True*, *callback=None*, *verbose=False*)
Bases: spynnaker.pyNN.models.neural_projections.connectors.abstract_connector.AbstractConnector

A connector that uses connection statistics based on the Small World network connectivity model.

Note: This is typically used from a population to itself.

Parameters

- **degree** (*float*) – the region length where nodes will be connected locally
- **rewiring** (*float*) – the probability of rewiring each edge
- **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.
- **n_connections** (*int or None*) – if specified, the number of efferent synaptic connections per neuron
- **rng** (*NumpyRNG or None*) – Seeded random number generator, or `None` to make one when needed.
- **safe** (*bool*) – If `True`, check that weights and delays have valid values. If `False`, this check is skipped.
- **callback** (*callable*) – if given, a callable that display a progress bar on the terminal.

  Note: Not supported by sPyNNaker.

- **verbose** (*bool*) – Whether to output extra information about the connectivity to a CSV file

create_synaptic_block(*pre_slices*, *post_slices*, *pre_vertex_slice*, *post_vertex_slice*, *synapse_type*, *synapse_info*)

Create a synaptic block from the data.

Parameters

- **weights** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **pre_slices** (*list(Slice)*) –
- **post_slices** (*list(Slice)*) –
- **pre_vertex_slice** (*Slice*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_type** (`AbstractSynapseType`) –
- **synapse_info** (`SynapseInformation`) –

Returns The synaptic matrix data to go to the machine, as a Numpy array

Return type ndarray

get_delay_maximum(*synapse_info*)

Get the maximum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (`SynapseInformation`) – the synapse info

Return type int or None

get_delay_minimum(*synapse_info*)

Get the minimum delay specified by the user in ms, or None if unbounded.

Parameters **synapse_info** (`SynapseInformation`) –

Return type int or None

get_n_connections_from_pre_vertex_maximum(*post_vertex_slice*, *synapse_info*, *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).

Parameters

- **delays** (*ndarray or NumpyRNG or int or float or list(int) or list(float)*) –
- **post_vertex_slice** (*Slice*) –
- **synapse_info** (SynapseInformation) –
- **min_delay** (*int or None*) –
- **max_delay** (*int or None*) –

Return type int

get_n_connections_to_post_vertex_maximum(*synapse_info*)

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

Parameters **synapse_info** (SynapseInformation) –

Return type int

get_weight_maximum(*synapse_info*)

Get the maximum of the weights for this connection.

Parameters **synapse_info** (SynapseInformation) –

Return type float

set_projection_information(*machine_time_step*, *synapse_info*)

sets a connectors projection info :param int machine_time_step: machine time step :param SynapseInformation synapse_info: the synapse info

spynnaker8.**StaticSynapse**

alias of spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_static.SynapseDynamicsStatic

spynnaker8.**STDPMechanism**

alias of spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_stdp.SynapseDynamicsSTDP

spynnaker8.**AdditiveWeightDependence**

alias of spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_additive.WeightDependenceAdditive

spynnaker8.**MultiplicativeWeightDependence**

alias of spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_multiplicative.WeightDependenceMultiplicative

spynnaker8.**SpikePairRule**

alias of spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spike_pair.TimingDependenceSpikePair

spynnaker8.**StructuralMechanismStatic**
alias of spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_static.SynapseDynamicsStructuralStatic

spynnaker8.**StructuralMechanismSTDP**
alias of spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_stdp.SynapseDynamicsStructuralSTDP

class spynnaker8.**LastNeuronSelection**(*spike_buffer_size=64*)
Bases: spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.partner_selection.abstract_partner_selection.AbstractPartnerSelection

Partner selection that picks a random source neuron from the neurons that spiked in the last timestep

Parameters **spike_buffer_size** – The size of the buffer for holding spikes

get_parameter_names()
Return the names of the parameters supported by this rule

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

Return type str

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*)
Write the parameters of the rule to the spec

Parameters **spec** (*DataSpecificationGenerator*) –

class spynnaker8.**RandomSelection**
Bases: spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.partner_selection.abstract_partner_selection.AbstractPartnerSelection

Partner selection that picks a random source neuron from all sources

get_parameter_names()
Return the names of the parameters supported by this rule

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

Return type str

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*)
Write the parameters of the rule to the spec

Parameters **spec** (*DataSpecificationGenerator*) –

class spynnaker8.**DistanceDependentFormation**(*grid=(16, 16)*, *p_form_forward=0.16*, *sigma_form_forward=2.5*, *p_form_lateral=1.0*, *sigma_form_lateral=1.0*)

Bases: `spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.formation.abstract_formation.AbstractFormation`

Formation rule that depends on the physical distance between neurons

Parameters

- **grid** (*tuple(int, int) or list(int) or ndarray(int)*) – (x, y) dimensions of the grid of distance
- **p_form_forward** (*float*) – The peak probability of formation on feed-forward connections
- **sigma_form_forward** (*float*) – The spread of probability with distance of formation on feed-forward connections
- **p_form_lateral** (*float*) – The peak probability of formation on lateral connections
- **sigma_form_lateral** (*float*) – The spread of probability with distance of formation on lateral connections

distance(*x0*, *x1*, *metric*)

Compute the distance between points x0 and x1 place on the grid using periodic boundary conditions.

Parameters

- **x0** (*ndarray(int)*) – first point in space
- **x1** (*ndarray(int)*) – second point in space
- **grid** (*ndarray(int)*) – shape of grid
- **metric** (*str*) – distance metric, i.e. `euclidian` or `manhattan` or `equidistant`

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 ndarray(float)

get_parameter_names()

Return the names of the parameters supported by this rule

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()

Get the amount of SDRAM used by the parameters of this rule

Return type int

vertex_executable_suffix

The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*)
Write the parameters of the rule to the spec

Parameters **spec** (*DataSpecificationGenerator*) –

class spynnaker8.**RandomByWeightElimination**(*threshold*, *prob_elim_depressed=0.0245*, *prob_elim_potentiated=0.00013600000000000003*)
Bases: `spynnaker.pyNN.models.neuron.structural_plasticity.synaptogenesis.elimination.abstract_elimination.AbstractElimination`

Elimination Rule that depends on the weight of a synapse

Parameters

- **threshold** (*float*) – Below this weight is considered depression, above or equal to this weight is considered potentiation (or the static weight of the connection on static weight connections)
- **prob_elim_depressed** (*float*) – The probability of elimination if the weight has been depressed (ignored on static weight connections)
- **prob_elim_potentiated** (*float*) – The probability of elimination of the weight has been potentiated or has not changed (and also used on static weight connections)

get_parameter_names()
Return the names of the parameters supported by this rule

Return type iterable(str)

get_parameters_sdram_usage_in_bytes()
Get the amount of SDRAM used by the parameters of this rule

Return type int

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

Return type str

write_parameters(*spec*, *weight_scale*)
Write the parameters of the rule to the spec

Parameters

- **spec** (*DataSpecificationGenerator*) –
- **weight_scale** (*float*) –

spynnaker8.**IF_cond_exp**
alias of `spynnaker.pyNN.models.neuron.builds.if_cond_exp_base.IFCondExpBase`

spynnaker8.**IF_curr_exp**
alias of `spynnaker.pyNN.models.neuron.builds.if_curr_exp_base.IFCurrExpBase`

spynnaker8.**IF_curr_alpha**
alias of `spynnaker.pyNN.models.neuron.builds.if_curr_alpha.IFCurrAlpha`

spynnaker8.**IF_curr_delta**
alias of `spynnaker.pyNN.models.neuron.builds.if_curr_delta.IFCurrDelta`

spynnaker8.**Izhikevich**
alias of `spynnaker.pyNN.models.neuron.builds.izk_curr_exp_base.IzkCurrExpBase`

class spynnaker8.**SpikeSourceArray**(*spike_times=None*)
Bases: *spynnaker.pyNN.models.abstract_pynn_model.AbstractPyNNModel*

create_vertex(*n_neurons*, *label*, *constraints*, *splitter*)
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 (AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type ApplicationVertex

default_population_parameters = {'splitter': None}

class spynnaker8.**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*, *splitter*)
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 (AbstractConstraint) or None*) – A list of constraints to give to the vertex, or None

Returns An application vertex for the population

Return type ApplicationVertex

default_population_parameters = {'max_rate': None, 'seed': None, 'splitter': 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

class spynnaker8.**Assembly**(**args*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

A group of neurons, may be heterogeneous, in contrast to a Population where all the neurons are of the same type.

Parameters

- **populations** (Population *or* PopulationView) – the populations or views to form the assembly out of
- **kwargs** – may contain *label* (a string describing the assembly)

class spynnaker8.**Population**(*size*, *cellclass*, *cellparams=None*, *structure=None*, *initial_values=None*, *label=None*, *constraints=None*, *additional_parameters=None*)

Bases: `spynnaker.pyNN.models.populations.population_base.PopulationBase`

PyNN 0.9 population object.

Parameters

- **size** (*int*) – The number of neurons in the population
- **cellclass** (*type or AbstractPyNNModel*) – The implementation of the individual neurons.
- **cellparams** (*dict(str,object) or None*) – Parameters to pass to `cellclass` if it is a class to instantiate. Must be `None` if `cellclass` is an instantiated object.
- **structure** (*BaseStructure*) –
- **initial_values** (*dict(str,float)*) – Initial values of state variables
- **label** (*str*) – A label for the population
- **constraints** (*list(AbstractConstraint)*) – Any constraints on how the population is deployed to SpiNNaker.
- **additional_parameters** (*dict(str, ..)*) – Additional parameters to pass to the vertex creation function.

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 MPI nodes.

Return type iterable(*IDMixin*)

all_cells

Return type list(*IDMixin*)

annotations

The annotations given by the end user

Return type dict(str, ..)

can_record(*variable*)

Determine whether *variable* can be recorded from this population.

Parameters **variable** (*str*) – The variable to answer the question about

Return type bool

celltype

Implements the PyNN expected celltype property

Returns The celltype this property has been set to

Return type *AbstractPyNNModel*

conductance_based
True if the population uses conductance inputs

Return type bool

static create(*cellclass*, *cellparams=None*, *n=1*)
Pass through method to the constructor defined by PyNN. Create `n` cells all of the same type.

Parameters

- **cellclass** (*type or* AbstractPyNNModel) – see `__init__()`
- **cellparams** (*dict* (*str*, *object*) *or None*) – see `__init__()`
- **n** (*int*) – see `__init__()` (`size` parameter)

Returns A New Population

Return type *Population*

describe(*template='population_default.txt'*, *engine='default'*)
Returns a human-readable description of the population.

The output may be customized 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.

Parameters

- **template** (*str*) – Template filename
- **engine** (*str or TemplateEngine or None*) – Template substitution engine

Return type str or dict

find_units(*variable*)
Get the units of a variable

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

Returns The units of the variable

Return type str

first_id
The ID of the first member of the population.

Return type int

get(*parameter_names*, *gather=True*, *simplify=True*)
Get the values of a parameter for every local cell in the population.

Parameters

- **parameter_names** (*str or iterable* (*str*)) – Name of parameter. This is either a single string or a list of strings
- **gather** (*bool*) – pointless on sPyNNaker
- **simplify** (*bool*) – ignored

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_data(*variables='all'*, *gather=True*, *clear=False*, *annotations=None*)
Return a Neo Block containing the data (spikes, state variables) recorded from the Assembly.

Parameters

- **variables** (*str or list(str)*) – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** (*bool*) – Whether to collect data from all MPI nodes or just the current node.

Note: This is irrelevant on sPyNNaker, which always behaves as if this parameter is True.

- **clear** (*bool*) – Whether recorded data will be deleted from the `Assembly`.
- **annotations** (*dict(str, ..)*) – annotations to put on the neo block

Return type Block

Raises **ConfigurationException** – If the variable or variables have not been previously set to record.

get_data_by_indexes(*variables*, *indexes*, *clear=False*, *annotations=None*)
Return a Neo *Block* containing the data (spikes, state variables) recorded from the Assembly.

Parameters

- **variables** (*str or list(str)*) – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **indexes** (*list(int)*) – List of neuron indexes to include in the data. Clearly only neurons recording will actually have any data. If None will be taken as all recording as in `get_data()`
- **clear** (*bool*) – Whether recorded data will be deleted.
- **annotations** (*dict(str, ..)*) – annotations to put on the neo block

Return type Block

Raises **ConfigurationException** – If the variable or variables have not been previously set to record.

get_initial_value(*variable*, *selector=None*)
Deprecated since version 6.0: Use `initial_values()` instead.

get_spike_counts(*gather=True*)
Return the number of spikes for each neuron.

Return type ndarray

id_to_index(*id*)
Given the ID(s) of cell(s) in the Population, return its (their) index (order in the Population).

Defined by http://neuralensemble.org/docs/PyNN/reference/populations.html

Parameters **id** (*int or iterable(int)*) –

Return type int or iterable(int)

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.

Defined by http://neuralensemble.org/docs/PyNN/reference/populations.html

Parameters **cell_id** (*int or iterable(int)*) –

Return type int or iterable(int)

index_to_id(*index*)
Given the index (order in the Population) of cell(s) in the Population, return their ID(s)

Parameters **index** (*int or iterable(int)*) –

Return type int or iterable(int)

initial_values

Return type dict

initialize(***kwargs*)
Set initial values of state variables, e.g. the membrane potential. Values passed to `initialize()` may be:

- single numeric values (all neurons set to the same value), or
- `RandomDistribution` objects, or
- lists / arrays of numbers of the same size as the population mapping functions, where a mapping function accepts a single argument (the cell index) and returns a single number.

Values should be expressed in the standard PyNN units (i.e. millivolts, nanoamps, milliseconds, microsiemens, nanofarads, event per second).

Examples:

```
p.initialize(v=-70.0)
p.initialize(v=rand_distr, gsyn_exc=0.0)
p.initialize(v=lambda i: -65 + i / 10.0)
```

inject(*current_source*)
Connect a current source to all cells in the Population.

Defined by http://neuralensemble.org/docs/PyNN/reference/populations.html

label
The label of the population

Return type str

last_id
The ID of the last member of the population.

Return type int

local_size
The number of local cells

Defined by http://neuralensemble.org/docs/PyNN/reference/populations.html

mark_no_changes()
Mark this population as not having changes to be mapped.

position_generator

Return type callable((int), ndarray)

positions
Return the position array for structured populations.

Returns a 2D array, one row per cell. Each row is three long, for X,Y,Z

Return type ndarray

record(*variables*, *to_file=None*, *sampling_interval=None*)
Record the specified variable or variables for all cells in the Population or view.

Parameters

- **variables** (`str or list(str)`) – either a single variable name or a list of variable names. For a given celltype class, `celltype.recordable` contains a list of variables that can be recorded for that celltype.
- **to_file** (`io or rawio or str`) – a file to automatically record to (optional). `write_data()` will be automatically called when *sim.end()* is called.
- **sampling_interval** (`int`) – a value in milliseconds, and an integer multiple of the simulation timestep.

requires_mapping
Whether this population requires mapping.

Return type bool

sample(*n*, *rng=None*)
Randomly sample *n* cells from the Population, and return a PopulationView object.

Parameters

- **n** (`int`) – The number of cells to put in the view.
- **rng** (`NumpyRNG`) – The random number generator to use

Return type *PopulationView*

set(***parameters*)
Set parameters of this population.

Parameters **parameters** – The parameters 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_by_selector(1, "tau_m", 20.0).
p.set_by_selector(1, {'tau_m':20, 'v_rest':-65})
```

Parameters

- **selector** – See `RangedList.set_value_by_selector()` as this is just a pass through method
- **parameter** (`str or dict(str, int or float or list(int) or list(float))`) – the parameter to set or dictionary of parameters to set
- **value** (`int or float or list(int) or list(float)`) – 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.

Parameters **constraint** (*AbstractConstraint*) –

set_initial_value (*variable*, *value*, *selector=None*)
Deprecated since version 6.0: Use `initialize()` instead.

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** (*int*) – the new value for the max atoms per core.

size
The number of neurons in the population

Return type int

spinnaker_get_data (*variable*)
Public accessor for getting data as a numpy array, instead of the neo based object

Parameters **variable** (*str or list (str)*) – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.

Returns array of the data

Return type ndarray

structure
Return the structure for the population.

Return type BaseStructure or None

tset (***kwargs*)
Deprecated since version 5.0: Use `set(parametername=value_array)` instead.

write_data (*io*, *variables='all'*, *gather=True*, *clear=False*, *annotations=None*)
Write recorded data to file, using one of the file formats supported by Neo.

Parameters

- **io** (*neo.io.baseio.BaseIO or str*) – a Neo IO instance, or a string for where to put a neo instance
- **variables** (*str or list (str)*) – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** (*bool*) – Whether to bring all relevant data together.

Note: SpiNNaker always gathers.

- **clear** (*bool*) – clears the storage data if set to true after reading it back
- **annotations** (*dict (str, ..)*) – annotations to put on the neo block

Raises **ConfigurationException** – If the variable or variables have not been previously set to record.

class spynnaker8.**PopulationView**(*parent*, *selector*, *label=None*)

Bases: spynnaker.pyNN.models.populations.population_base.PopulationBase

A view of a subset of neurons within a *Population*.

In most ways, Populations and PopulationViews have the same behaviour, i.e., they can be recorded, connected with Projections, etc. It should be noted that any changes to neurons in a PopulationView will be reflected in the parent Population and *vice versa.*

It is possible to have views of views.

Note: Selector to Id is actually handled by AbstractSized.

Parameters

- **parent** (Population *or* PopulationView) – the population or view to make the view from
- **selector** (*None or slice or int or list(bool) or list(int) or ndarray(bool) or ndarray(int)*) – a slice or numpy mask array. The mask array should either be a boolean array (ideally) of the same size as the parent, or an integer array containing cell indices, i.e. if *p.size == 5* then:

```
PopulationView(p, array([False, False, True, False, True]))
PopulationView(p, array([2, 4]))
PopulationView(p, slice(2, 5, 2))
```

  will all create the same view.
- **label** (*str*) – A label for the view

all()

Iterator over cell IDs (on all MPI nodes).

Return type iterable(IDMixin)

all_cells

An array containing the cell IDs of all neurons in the Population (all MPI nodes).

Return type list(*IDMixin*)

can_record(*variable*)

Determine whether variable can be recorded from this population.

Return type bool

celltype

The type of neurons making up the underlying Population.

Return type *AbstractPyNNModel*

conductance_based

Indicates whether the post-synaptic response is modelled as a change in conductance or a change in current.

Return type bool

describe(*template='populationview_default.txt'*, *engine='default'*)

Returns a human-readable description of the population view.

The output may be customized 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.

Parameters

- **`template`** (*`str`*) – Template filename
- **`engine`** (*`str or TemplateEngine or None`*) – Template substitution engine

Return type str or dict

`find_units` (*variable*)

Get the units of a variable

> **Warning:** No PyNN description of this method.

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

Returns The units of the variable

Return type str

`get` (*parameter_names*, *gather=False*, *simplify=True*)

Get the values of the given parameters for every local cell in the population, or, if `gather=True`, for all cells in the population.

Values will be expressed in the standard PyNN units (i.e. millivolts, nanoamps, milliseconds, microsiemens, nanofarads, event per second).

> **Note:** SpiNNaker always gathers.

Parameters

- **`parameter_names`** (*`str or list(str)`*) –
- **`gather`** (*`bool`*) –
- **`simplify`** (*`bool`*) –

Return type iterable(float)

`get_data` (*variables='all'*, *gather=True*, *clear=False*, *annotations=None*)

Return a Neo Block containing the data(spikes, state variables) recorded from the Population.

Parameters

- **`variables`** (*`str or list(str)`*) – Either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **`gather`** (*`bool`*) – For parallel simulators, if gather is True, all data will be gathered to all nodes and the Neo Block will contain data from all nodes. Otherwise, the Neo Block will contain only data from the cells simulated on the local node.

  > **Note:** SpiNNaker always gathers.

- **`clear`** (*`bool`*) – If True, recorded data will be deleted from the Population.
- **`annotations`** (*`dict(str, ..)`*) – annotations to put on the neo block

Return type Block

Raises **ConfigurationException** – If the variable or variables have not been previously set to record.

get_spike_counts(*gather=True*)
Returns a dict containing the number of spikes for each neuron.

The dict keys are neuron IDs, not indices.

Note: Implementation of this method is different to Population as the Populations uses PyNN 7 version of the `get_spikes` method which does not support indexes.

Parameters **gather** (*bool*) –

Note: SpiNNaker always gathers.

Return type dict(int,int)

grandparent
Returns the parent Population at the root of the tree (since the immediate parent may itself be a PopulationView).

The name "grandparent" is of course a little misleading, as it could be just the parent, or the great, great, great, . . . , grandparent.

Return type *Population*

id_to_index(*id*)
Given the ID(s) of cell(s) in the PopulationView, return its / their index / indices(order in the PopulationView).

assert pv.id_to_index(pv[3]) == 3

Parameters **id** (*int or list(int)*) –

Return type int or list(int)

index_in_grandparent(*indices*)
Given an array of indices, return the indices in the parent population at the root of the tree.

Parameters **indices** (*list(int)*) –

Return type list(int)

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

Return type dict(str, ..)

initialize(***initial_values*)
Set initial values of state variables, e.g. the membrane potential. Values passed to `initialize()` may be:

- single numeric values (all neurons set to the same value), or
- `RandomDistribution` objects, or
- lists / arrays of numbers of the same size as the population mapping functions, where a mapping function accepts a single argument (the cell index) and returns a single number.

Values should be expressed in the standard PyNN units (i.e. millivolts, nanoamps, milliseconds, microsiemens, nanofarads, events per second).

Examples:

```
p.initialize(v=-70.0)
p.initialize(v=rand_distr, gsyn_exc=0.0)
p.initialize(v=lambda i: -65 + i / 10.0)
```

label
A label for the Population View.

Return type str

mask
The selector mask that was used to create this view.

Return type None or slice or int or list(bool) or list(int) or ndarray(bool) or ndarray(int)

parent
A reference to the parent Population (that this is a view of).

Return type *Population*

record(*variables*, *to_file=None*, *sampling_interval=None*)
Record the specified variable or variables for all cells in the Population or view.

Parameters

- **variables** (*str or list(str)*) – either a single variable name, or a list of variable names, or `all` to record everything. For a given celltype class, celltype.recordable contains a list of variables that can be recorded for that celltype.
- **to_file** (*io or rawio or str*) – If specified, should be a Neo IO instance and *write_data()* will be automatically called when *sim.end()* is called.
- **sampling_interval** (*int*) – should be a value in milliseconds, and an integer multiple of the simulation timestep.

sample(*n*, *rng=None*)
Randomly sample *n* cells from the Population view, and return a new PopulationView object.

Parameters

- **n** (*int*) – The number of cells to select
- **rng** (*NumpyRNG*) – Random number generator

Return type *PopulationView*

set(***parameters*)
Set one or more parameters for every cell in the population. Values passed to *set()* may be:

- single values,
- `RandomDistribution` objects, or
- lists / arrays of values of the same size as the population mapping functions, where a mapping function accepts a single argument (the cell index) and returns a single value.

Here, a "single value" may be either a single number or a list / array of numbers (e.g. for spike times).

Values should be expressed in the standard PyNN units (i.e. millivolts, nanoamps, milliseconds, microsiemens, nanofarads, event per second).

Examples:

```
p.set(tau_m=20.0, v_rest=-65).
p.set(spike_times=[0.3, 0.7, 0.9, 1.4])
p.set(cm=rand_distr, tau_m=lambda i: 10 + i / 10.0)
```

size
The total number of neurons in the Population View.

Return type int

write_data(*io*, *variables='all'*, *gather=True*, *clear=False*, *annotations=None*)
Write recorded data to file, using one of the file formats supported by Neo.

Parameters

- **io** (*neo.io.BaseIO or str*) – a Neo IO instance or the name of a file to write
- **variables** (*str or list(str)*) – either a single variable name or a list of variable names. These must have been previously recorded, otherwise an Exception will be raised.
- **gather** (*bool*) – For parallel simulators, if this is True, all data will be gathered to the master node and a single output file created there. Otherwise, a file will be written on each node, containing only data from the cells simulated on that node.

  Note: SpiNNaker always gathers.

- **clear** (*bool*) – If this is True, recorded data will be deleted from the Population.
- **annotations** (*dict(str, ..)*) – should be a dict containing simple data types such as numbers and strings. The contents will be written into the output data file as metadata.

Raises **ConfigurationException** – If the variable or variables have not been previously set to record.

spynnaker8.**SpiNNakerProjection**
alias of *spynnaker.pyNN.models.projection.Projection*

spynnaker8.**end**(*_=True*)
Cleans up the SpiNNaker machine and software

Parameters _ – was named compatible_output, which we don't care about, so is a non-existent parameter

spynnaker8.**setup**(*timestep=<sphinx.ext.autodoc.importer._MockObject object>*, *min_delay=<sphinx.ext.autodoc.importer._MockObject object>*, *max_delay=<sphinx.ext.autodoc.importer._MockObject object>*, *graph_label=None*, *database_socket_addresses=None*, *extra_algorithm_xml_paths=None*, *extra_mapping_inputs=None*, *extra_mapping_algorithms=None*, *extra_pre_run_algorithms=None*, *extra_post_run_algorithms=None*, *extra_load_algorithms=None*, *time_scale_factor=None*, *n_chips_required=None*, *n_boards_required=None*, ***extra_params*)
The main method needed to be called to make the PyNN 0.8 setup. Needs to be called before any other function

Parameters

- **timestep** (*float*) – the time step of the simulations
- **min_delay** (*float or str*) – the min delay of the simulation
- **max_delay** (*float or str*) – the max delay of the simulation
- **graph_label** (*str or None*) – the label for the graph

- **database_socket_addresses** (*iterable(SocketAddress)*) – the sockets used by external devices for the database notification protocol
- **extra_algorithm_xml_paths** (*list(str) or None*) – list of paths to where other XML are located
- **extra_mapping_inputs** (*dict(str, Any) or None*) – other inputs used by the mapping process
- **extra_mapping_algorithms** (*list(str) or None*) – other algorithms to be used by the mapping process
- **extra_pre_run_algorithms** (*list(str) or None*) – extra algorithms to use before a run
- **extra_post_run_algorithms** (*list(str) or None*) – extra algorithms to use after a run
- **extra_load_algorithms** (*list(str) or None*) – extra algorithms to use within the loading phase
- **time_scale_factor** (*int or None*) – multiplicative factor to the machine time step (does not affect the neuron models accuracy)
- **n_chips_required** (*int or None*) – Deprecated! Use n_boards_required instead. Must be None if n_boards_required specified.
- **n_boards_required** (*int or None*) – if you need to be allocated a machine (for spalloc) before building your graph, then fill this in with a general idea of the number of boards you need so that the spalloc system can allocate you a machine big enough for your needs.
- **extra_params** – other keyword argumets used to configure PyNN

Returns MPI rank (always 0 on SpiNNaker)

Return type int

Raises ConfigurationException – if both `n_chips_required` and `n_boards_required` are used.

`spynnaker8.`**run**(*simtime*, *callbacks=None*)

The run() function advances the simulation for a given number of milliseconds, e.g.:

Parameters

- **simtime** (*float*) – time to run for (in milliseconds)
- **callbacks** – callbacks to run

Returns the actual simulation time that the simulation stopped at

Return type float

`spynnaker8.`**run_until**(*tstop*)

Run until a (simulation) time period has completed.

Parameters tstop (*float*) – the time to stop at (in milliseconds)

Returns the actual simulation time that the simulation stopped at

Return type float

`spynnaker8.`**run_for**(*simtime*, *callbacks=None*)

The run() function advances the simulation for a given number of milliseconds, e.g.:

Parameters

- **simtime** (*float*) – time to run for (in milliseconds)
- **callbacks** – callbacks to run

Returns the actual simulation time that the simulation stopped at

Return type float

`spynnaker8.num_processes()`

The number of MPI processes.

Note: Always 1 on SpiNNaker, which doesn't use MPI.

Returns the number of MPI processes

Return type int

`spynnaker8.rank()`

The MPI rank of the current node.

Note: Always 0 on SpiNNaker, which doesn't use MPI.

Returns MPI rank

Return type int

`spynnaker8.reset(annotations=None)`

Resets the simulation to t = 0

Parameters **annotations** (*dict(str, ..)*) – the annotations to the data objects

Return type None

`spynnaker8.set_number_of_neurons_per_core(neuron_type, max_permitted)`

Sets a ceiling on the number of neurons of a given type that can be placed on a single core.

Parameters

- **neuron_type** (*type(AbstractPopulationVertex)*) – neuron type
- **max_permitted** (*int*) – the number to set to

`spynnaker8.get_projections_data(projection_data)`

Parameters **projection_data** (*dict(Projection, list(int) or tuple(int) or None)*) – the projection to attributes mapping

Returns a extracted data object with get method for getting the data

Return type *ExtractedData*

`spynnaker8.Projection(presynaptic_population, postsynaptic_population, connector, synapse_type=None, source=None, receptor_type='excitatory', space=None, label=None)`

Used to support PEP 8 spelling correctly

Parameters

- **presynaptic_population** (Population) – the source pop

- **postsynaptic_population** (`Population`) – the dest pop
- **connector** (`AbstractConnector`) – the connector type
- **synapse_type** (`AbstractStaticSynapseDynamics`) – the synapse type
- **source** (*None*) – Unsupported; must be `None`
- **receptor_type** (*str*) – the receptor type
- **space** (*Space or None*) – the space object
- **label** (*str or None*) – the label

Returns a projection object for SpiNNaker

Return type *Projection*

`spynnaker8.`**`get_current_time`**`()`

Gets the time within the simulation

Returns returns the current time

`spynnaker8.`**`create`**`(`*cellclass*, *cellparams=None*, *n=1*`)`

Builds a population with certain params

Parameters

- **cellclass** (*type or* `AbstractPyNNModel`) – population class
- **cellparams** – population params.
- **n** (*int*) – n neurons

Return type *Population*

`spynnaker8.`**`connect`**`(`*pre*, *post*, *weight=0.0*, *delay=None*, *receptor_type=None*, *p=1*, *rng=None*`)`

Builds a projection

Parameters

- **pre** (`Population`) – source pop
- **post** (`Population`) – destination pop
- **weight** (*float*) – weight of the connections
- **delay** (*float*) – the delay of the connections
- **receptor_type** (*str*) – excitatory / inhibitory
- **p** (*float*) – probability
- **rng** (*NumpyRNG*) – random number generator

`spynnaker8.`**`get_time_step`**`()`

The integration time step

Returns get the time step of the simulation (in ms)

Return type float

`spynnaker8.`**`get_min_delay`**`()`

The minimum allowed synaptic delay; delays will be clamped to be at least this.

Returns returns the min delay of the simulation

Return type int

`spynnaker8.get_max_delay()`
The maximum allowed synaptic delay; delays will be clamped to be at most this.

Returns returns the max delay of the simulation

Return type int

`spynnaker8.initialize(cells, **initial_values)`
Sets cells to be initialised to the given values

Parameters

- **cells** (Population or PopulationView) – the cells to change params on
- **initial_values** – the params and their values to change

`spynnaker8.list_standard_models()`
Return a list of all the StandardCellType classes available for this simulator.

Return type list(str)

`spynnaker8.name()`
Returns the name of the simulator

Return type str

`spynnaker8.record(variables, source, filename, sampling_interval=None, annotations=None)`
Sets variables to be recorded.

Parameters

- **variables** (str or list(str)) – may be either a single variable name or a list of variable names. For a given celltype class, celltype.recordable contains a list of variables that can be recorded for that celltype.
- **source** (Population or PopulationView) – where to record from
- **filename** (str) – file name to write data to
- **sampling_interval** – how often to sample the recording, not ignored so far
- **annotations** (dict(str, ..)) – the annotations to data writers

Returns neo object

Return type Block

`spynnaker8.record_v(source, filename)`
Deprecated method for getting voltage. This is not documented in the public facing API.

Deprecated since version 5.0.

Parameters

- **source** (Population or PopulationView) – the population / view / assembly to record
- **filename** (str) – the neo file to write to

Return type None

`spynnaker8.record_gsyn(source, filename)`
Deprecated method for getting both types of gsyn. This is not documented in the public facing API

Deprecated since version 5.0.

Parameters

- **source** (Population *or* PopulationView) – the population / view / assembly to record
- **filename** (*str*) – the neo file to write to

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W