sPyNNaker8 Documentation

Jan 29, 2020

Contents

These pages document the python code for the sPyNNaker8 module which is part of the SpiNNaker Project.

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

Contents:

CHAPTER 1

spynnaker8 package

1.1 Subpackages

1.1.1 spynnaker8.external_devices package

1.1.1.1 Module contents

The spynnaker.pyNN package contains the front end specifications and implementation for the PyNN High-level API (http://neuralensemble.org/trac/PyNN)

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

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

DOWN_POLARITY = 'DOWN'

MERGED_POLARITY = 'MERGED'

MODE_128 = '128'

MODE_16 = '16'

MODE_32 = '32'

MODE_64 = '64'

UP_POLARITY = 'UP'

static get_n_neurons(*mode*, *polarity*)

get_outgoing_partition_constraints(*partition*)

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

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

Returns A list of constraints

Return type list(AbstractConstraint)

pause_stop_commands

The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

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

Return type iterable(MultiCastCommand)

class spynnaker8.external_devices.**MunichRetinaDevice**(*retina_key*, *spinnaker_link_id*, *position*, *label=None*, *polarity=None*, *board_address=None*)

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

DOWN_POLARITY = 'DOWN'

LEFT_RETINA = 'LEFT'

LEFT_RETINA_DISABLE = 69

LEFT_RETINA_ENABLE = 69

LEFT_RETINA_KEY_SET = 67

MANAGEMENT_BIT = 1024

MANAGEMENT_MASK = 4294965248

MERGED_POLARITY = 'MERGED'

RIGHT_RETINA = 'RIGHT'

RIGHT_RETINA_DISABLE = 70

RIGHT_RETINA_ENABLE = 70

RIGHT_RETINA_KEY_SET = 68

UP_POLARITY = 'UP'

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

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

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

Returns A list of constraints

Return type list(AbstractConstraint)

pause_stop_commands
The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

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

Return type iterable(MultiCastCommand)

class 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.application_vertex.ApplicationVertex, spinn_front_end_common.abstract_models.abstract_vertex_with_dependent_vertices.AbstractVertexWithEdgeToDependentVertices, spinn_front_end_common.abstract_models.abstract_generates_data_specification.AbstractGeneratesDataSpecification, spinn_front_end_common.abstract_models.abstract_has_associated_binary.AbstractHasAssociatedBinary, spinn_front_end_common.abstract_models.abstract_provides_outgoing_partition_constraints.AbstractProvidesOutgoingPartitionConstraints, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

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

PARAMS_REGION = 1

PARAMS_SIZE = 28

SYSTEM_REGION = 0

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

Parameters

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

default_initial_values = {}

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

dependent_vertices()
Return the vertices which this vertex depends upon

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

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

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

Return type iterable(str) Return the dependent edge identifier

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

Parameters

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

Return type None

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

Return type str

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

Return type ExecutableType

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

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

Returns A list of constraints

Return type list(AbstractConstraint)

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

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

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

Return type ResourceContainer

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

n_atoms
The number of atoms in the vertex

Return type int

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

class 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

class spynnaker8.external_devices.**PushBotRetinaViewer**(*resolution*, *port=0*, *display_max=33.0*, *frame_time_ms=10*, *decay_time_constant_ms=100*)

Bases: threading.Thread

local_host

local_port

run()

Method representing the thread's activity.

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

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*)

Create a vertex for a population of the model

Parameters

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

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

class 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** – The optional instance key to use
- **uart_id** – 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

master_slave_key

master_slave_set_master_clock_active(*time=None*)

master_slave_set_master_clock_not_started(*time=None*)

master_slave_set_slave(*time=None*)

master_slave_use_internal_counter(*time=None*)

mode

poll_individual_sensor_continuously(*sensor_id*, *time_in_ms*, *time=None*)

poll_individual_sensor_continuously_key

poll_sensors_once(*sensor_id*, *time=None*)

poll_sensors_once_key

protocol_instance = 0

push_bot_laser_config_active_time(*active_time*, *time=None*)

push_bot_laser_config_active_time_key

push_bot_laser_config_total_period(*total_period*, *time=None*)

push_bot_laser_config_total_period_key

push_bot_laser_set_frequency(*frequency*, *time=None*)

push_bot_laser_set_frequency_key

push_bot_led_back_active_time(*active_time*, *time=None*)

push_bot_led_back_active_time_key

push_bot_led_front_active_time(*active_time*, *time=None*)

push_bot_led_front_active_time_key

push_bot_led_set_frequency(*frequency*, *time=None*)

push_bot_led_set_frequency_key

push_bot_led_total_period(*total_period*, *time=None*)

push_bot_led_total_period_key

push_bot_motor_0_leaking_towards_zero(*velocity*, *time=None*)

push_bot_motor_0_leaking_towards_zero_key

push_bot_motor_0_permanent(*velocity*, *time=None*)

push_bot_motor_0_permanent_key

push_bot_motor_1_leaking_towards_zero(*velocity*, *time=None*)

push_bot_motor_1_leaking_towards_zero_key

push_bot_motor_1_permanent(*velocity*, *time=None*)

push_bot_motor_1_permanent_key

push_bot_speaker_config_active_time(*active_time*, *time=None*)

push_bot_speaker_config_active_time_key

push_bot_speaker_config_total_period(*total_period*, *time=None*)

push_bot_speaker_config_total_period_key

push_bot_speaker_set_melody(*melody*, *time=None*)

push_bot_speaker_set_melody_key

push_bot_speaker_set_tone(*frequency*, *time=None*)

push_bot_speaker_set_tone_key

pwm_pin_output_timer_a_channel_0_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_a_channel_0_ratio_key

pwm_pin_output_timer_a_channel_1_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_a_channel_1_ratio_key

pwm_pin_output_timer_a_duration(*timer_period*, *time=None*)

pwm_pin_output_timer_a_duration_key

pwm_pin_output_timer_b_channel_0_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_b_channel_0_ratio_key()

pwm_pin_output_timer_b_channel_1_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_b_channel_1_ratio_key

pwm_pin_output_timer_b_duration(*timer_period*, *time=None*)

pwm_pin_output_timer_b_duration_key

pwm_pin_output_timer_c_channel_0_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_c_channel_0_ratio_key

pwm_pin_output_timer_c_channel_1_ratio(*timer_period*, *time=None*)

pwm_pin_output_timer_c_channel_1_ratio_key()

pwm_pin_output_timer_c_duration(*timer_period*, *time=None*)

pwm_pin_output_timer_c_duration_key

query_state_of_io_lines(*time=None*)

query_state_of_io_lines_key

remove_payload_logic_to_current_output(*payload*, *time=None*)

remove_payload_logic_to_current_output_key

reset_retina(*time=None*)

reset_retina_key

sensor_transmission_key(*sensor_id*)

static sent_mode_command()
True if the mode command has ever been requested by any instance

set_mode(*time=None*)

set_mode_key

set_output_pattern_for_payload(*payload*, *time=None*)

set_output_pattern_for_payload_key

set_payload_pins_to_high_impedance(*payload*, *time=None*)

set_payload_pins_to_high_impedance_key

set_retina_key(*new_key*, *time=None*)

set_retina_key_key

set_retina_transmission(*retina_key=<RetinaKey.NATIVE_128_X_128: 67108864>*, *retina_payload=None*, *time=None*)
Set the retina transmission key

Parameters

- **retina_key** – the new key for the retina
- **retina_payload** (*enum or None*) – the new payload for the set retina key command packet
- **time** – when to transmit this packet

Returns the command to send

Return type spinn_front_end_common.utility_models.multi_cast_command.MultiCastCommand

set_retina_transmission_key

turn_off_sensor_reporting(*sensor_id*, *time=None*)

turn_off_sensor_reporting_key

uart_id

class spynnaker8.external_devices.**PushBotLaser**

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

An enumeration.

LASER_ACTIVE_TIME = 1

LASER_FREQUENCY = 2

LASER_TOTAL_PERIOD = 0

class spynnaker8.external_devices.**PushBotLED**

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

An enumeration.

LED_BACK_ACTIVE_TIME = 2

LED_FREQUENCY = 3

LED_FRONT_ACTIVE_TIME = 1

LED_TOTAL_PERIOD = 0

class spynnaker8.external_devices.**PushBotMotor**

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

An enumeration.

MOTOR_0_LEAKY = 1

MOTOR_0_PERMANENT = 0

MOTOR_1_LEAKY = 3

MOTOR_1_PERMANENT = 2

class spynnaker8.external_devices.**PushBotSpeaker**

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

An enumeration.

SPEAKER_ACTIVE_TIME = 1

SPEAKER_MELODY = 3

SPEAKER_TONE = 2

SPEAKER_TOTAL_PERIOD = 0

class spynnaker8.external_devices.**PushBotRetinaResolution**

Bases: enum.Enum

An enumeration.

DOWNSAMPLE_16_X_16 = <RetinaKey.DOWNSAMPLE_16_X_16: 268435456>

DOWNSAMPLE_32_X_32 = <RetinaKey.DOWNSAMPLE_32_X_32: 201326592>

DOWNSAMPLE_64_X_64 = <RetinaKey.DOWNSAMPLE_64_X_64: 134217728>

NATIVE_128_X_128 = <RetinaKey.NATIVE_128_X_128: 67108864>

class 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

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.push_bot_ethernet.push_bot_ethernet_device.PushBotEthernetDevice, spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

The Laser of a PushBot

Parameters

- **laser** – The PushBotLaser value to control
- **protocol** – The protocol instance to get commands from
- **start_active_time** – The "active time" value to send at the start
- **start_total_period** – The "total period" value to send at the start
- **start_frequency** – The "frequency" to send at the start
- **timesteps_between_send** – The number of timesteps between sending commands to the device, or None to use the default

pause_stop_commands

The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

set_command_protocol(*command_protocol*)

Set the protocol use to send setup and shutdown commands, separately from the protocol used to control the device

start_resume_commands

The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands

The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

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.push_bot_ethernet.push_bot_ethernet_device.PushBotEthernetDevice, spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

The LED of a PushBot

Parameters

- **led** – The PushBotLED parameter to control
- **protocol** – The protocol instance to get commands from
- **start_active_time_front** – The "active time" to set for the front LED at the start
- **start_active_time_back** – The "active time" to set for the back LED at the start
- **start_total_period** – The "total period" to set at the start
- **start_frequency** – The "frequency" to set at the start
- **timesteps_between_send** – The number of timesteps between sending commands to the device, or None to use the default

pause_stop_commands

The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

set_command_protocol(*command_protocol*)

Set the protocol use to send setup and shutdown commands, separately from the protocol used to control the device

start_resume_commands

The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

timed_commands

The commands to be sent at given times in the simulation

Return type iterable(MultiCastCommand)

class spynnaker8.external_devices.**PushBotEthernetMotorDevice**(*motor*, *protocol*, *timesteps_between_send=None*)

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

The motor of a PushBot

Parameters

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

pause_stop_commands
The commands needed when pausing or stopping simulation

Return type iterable(MultiCastCommand)

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

start_resume_commands
The commands needed when starting or resuming simulation

Return type iterable(MultiCastCommand)

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

Return type iterable(MultiCastCommand)

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.push_bot_ethernet.push_bot_ethernet_device.PushBotEthernetDevice, spinn_front_end_common.abstract_models.abstract_send_me_multicast_commands_vertex.AbstractSendMeMulticastCommandsVertex, spinn_front_end_common.abstract_models.impl.provides_key_to_atom_mapping_impl.ProvidesKeyToAtomMappingImpl

The Speaker of a PushBot

Parameters

- **speaker** – The PushBotSpeaker value to control
- **protocol** – 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

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

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

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

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

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

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

class 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

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.push_bot_ethernet.

push_bot_ethernet_laser_device.PushBotEthernetLaserDevice, pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex

The Laser of a PushBot

Parameters

- **laser** – The PushBotLaser value to control
- **protocol** – The protocol instance to get commands from
- **spinnaker_link_id** – The SpiNNakerLink that the PushBot is connected to
- **n_neurons** – The number of neurons in the device
- **label** – A label for the device
- **board_address** – 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.push_bot_ethernet.push_bot_ethernet_led_device.PushBotEthernetLEDDevice, pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex

The LED of a PushBot

Parameters

- **led** – The PushBotLED parameter to control
- **protocol** – The protocol instance to get commands from
- **spinnaker_link_id** – The SpiNNakerLink connected to
- **n_neurons** – The number of neurons in the device
- **label** – The label of the device
- **board_address** – The IP address of the board that the device is connected to
- **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

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.push_bot_ethernet.push_bot_ethernet_motor_device.PushBotEthernetMotorDevice, pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex

The motor of a PushBot

Parameters

- **motor** – a PushBotMotor value to indicate the motor to control
- **protocol** – The protocol used to control the device
- **spinnaker_link_id** – The SpiNNakerLink connected to
- **n_neurons** – The number of neurons in the device
- **label** – The label of the device
- **board_address** – 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.push_bot_ethernet.push_bot_ethernet_speaker_device.PushBotEthernetSpeakerDevice, pacman.model.graphs.application.application_spinnaker_link_vertex.ApplicationSpiNNakerLinkVertex

The speaker of a PushBot

Parameters

- **speaker** – The PushBotSpeaker value to control
- **protocol** – The protocol instance to get commands from
- **spinnaker_link_id** – The SpiNNakerLink connected to
- **n_neurons** – The number of neurons in the device
- **label** – The label of the device
- **board_address** – 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}

graph_mapper(*graph_mapper*)

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='LiveSpikeR*

Bases: spinn_front_end_common.utilities.connections.live_event_connection.LiveEventConnection

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

Parameters

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

send_spike(*label*, *neuron_id*, *send_full_keys=False*)

Send a spike from a single neuron

Parameters

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

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

Parameters

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

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 of 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** (*function(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** (function(str, SpynnakerLiveEventConnection) -> 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*)

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** (*function(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** – 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** (function(str, SpynnakerLiveEventConnection) -> 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** (function(str, SpynnakerLiveEventConnection) -> 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** – The label of the Population to set the rates of
- **neuron_id** – The neuron ID to set the rate of
- **rate** – 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** – The label of the Population to set the rates of
- **neuron_id_rates** – 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** (spynnaker.pyNN.models.pynn_population_common.PyNNPopulationCommon) – The population to activate the live output for
- **database_notify_host** (*str*) – The hostname for the device which is listening to the database notification.
- **database_ack_port_num** (*int*) – The port number to which a external device will acknowledge that they have finished reading the database and are ready for it to start execution
- **database_notify_port_num** (*int*) – The port number to which a external device will receive the database is ready command
- **board_address** (*str*) – A fixed board address required for the tag, or None if any address is OK
- **key_prefix** (*int or None*) – the prefix to be applied to the key
- **prefix_type** – if the prefix type is 32 bit or 16 bit
- **message_type** – If the message is a EIEIO command message, or an EIEIO data message with 16 bit or 32 bit keys.
- **payload_as_time_stamps** –
- **right_shift** –
- **use_payload_prefix** –
- **notify** –
- **payload_prefix** –
- **payload_right_shift** –
- **number_of_packets_sent_per_time_step** –
- **port** (*int*) – The UDP port to which the live spikes will be sent. If not specified, the port will be taken from the "live_spike_port" parameter in the "Recording" section of the sPyNNaker configuration file.

- **host** (*str*) – The host name or IP address to which the live spikes will be sent. If not specified, the host will be taken from the "live_spike_host" parameter in the "Recording" section of the sPyNNaker configuration file.
- **tag** (*int*) – The IP tag to be used for the spikes. If not specified, one will be automatically assigned
- **strip_sdp** (*bool*) – Determines if the SDP headers will be stripped from the transmitted packet.
- **use_prefix** (*bool*) – Determines if the spike packet will contain a common prefix for the spikes
- **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*)

Activate the output of spikes from a population to an external device. Note that all spikes will be sent to the device.

Parameters

- **population** (spynnaker.pyNN.models.pynn_population_common.PyNNPopulationCommon) – The pyNN population object from which spikes will be sent.
- **device** (spynnaker.pyNN.models.pynn_population_common.PyNNPopulationCommon or pacman.model.graphs.application.ApplicationVertex) – The pyNN population or external device to which the spikes will be sent.

spynnaker8.external_devices.**SpikeInjector**(*notify=True*, *database_notify_host=None*, *database_notify_port_num=None*, *database_ack_port_num=None*)

Supports adding a spike injector to the application graph.

Parameters

- **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

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** – hostname to connect to
- **notify_port** – port num for the notify command
- **ack_port** – port num for the acknowledge command

Return type None

spynnaker8.external_devices.**run_forever**()
Supports running forever in PyNN 0.8/0.9 format

Returns 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** (spynnaker.pyNN.models.pynn_population_common.PyNNPopulationCommon) – The population to control
- **control_label_extension** (*str*) – An extension to add to the label of the Poisson source. Must match up with the equivalent in the SpynnakerPoissonControlConnection
- **receive_port** (*int*) – The port that the SpiNNaker board should listen on
- **database_notify_host** (*str*) – the hostname for the device which is listening to the database notification.
- **database_ack_port_num** (*int*) – the port number to which a external device will acknowledge that they have finished reading the database and are ready for it to start execution
- **database_notify_port_num** (*int*) – The port number to which a external device will receive the database is ready command
- **reserve_reverse_ip_tag** (*bool*) – True if a reverse ip tag is to be used, False if SDP is to be used (default)

1.1.2 spynnaker8.extra_models package

1.1.2.1 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

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.

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.weight_dependence_additive_triplet.WeightDependenceAdditiveTriplet

spynnaker8.extra_models.**PfisterSpikeTriplet**
alias of *spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_pfister_spike_triplet.TimingDependencePfisterSpikeTriplet*

spynnaker8.extra_models.**SpikeNearestPairRule**
alias of *spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_spike_nearest_pair.TimingDependenceSpikeNearestPair*

spynnaker8.extra_models.**RecurrentRule**
alias of *spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_recurrent.TimingDependenceRecurrent*

spynnaker8.extra_models.**Vogels2011Rule**
alias of *spynnaker8.models.synapse_dynamics.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*)
Create a vertex for a population of the model

Parameters

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

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {'seed': None}

classmethod get_max_atoms_per_core()
Get the maximum number of atoms per core for this model

Return type int

classmethod set_model_max_atoms_per_core(*n_atoms=500*)
Set the maximum number of atoms per core for this model

Parameters **n_atoms** (*int or None*) – The new maximum, or None for the largest possible

1.1.3 spynnaker8.models package

1.1.3.1 Subpackages

spynnaker8.models.connectors package

Submodules

spynnaker8.models.connectors.all_to_all_connector module

class spynnaker8.models.connectors.all_to_all_connector.**AllToAllConnector**(*allow_self_connections=True*, *safe=True*, *verbose=None*, *callbacks=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.all_to_all_connector.AllToAllConnector, pyNN.connectors.AllToAllConnector

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** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **verbose** –
- **callbacks** –

spynnaker8.models.connectors.array_connector module

class spynnaker8.models.connectors.array_connector.**ArrayConnector**(*array*, *safe=True*, *callback=None*, *verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.array_connector.ArrayConnector

Create an array connector.

Parameters **array** (*integer*) – an array of integers

spynnaker8.models.connectors.csa_connector module

class spynnaker8.models.connectors.csa_connector.**CSAConnector**(*cset*, *safe=True*, *callback=None*, *verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.csa_connector.CSAConnector

Create an CSA (Connection Set Algebra, Djurfeldt 2012) connector.

Parameters **cset** (*string*) – a connection set description

spynnaker8.models.connectors.distance_dependent_probability_connector module

class spynnaker8.models.connectors.distance_dependent_probability_connector.**DistanceDepende**

Bases: spynnaker.pyNN.models.neural_projections.connectors.distance_dependent_probability_connector.DistanceDependentProbabilityConnector, pyNN.connectors.DistanceDependentProbabilityConnector

Make connections using a distribution which varies with distance.

Parameters

- **d_expression** (*string*) – 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.
- **n_connections** (*int*) – The number of efferent synaptic connections per neuron.
- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.

spynnaker8.models.connectors.fixed_number_post_connector module

class spynnaker8.models.connectors.fixed_number_post_connector.**FixedNumberPostConnector**(*n*, *al-*
low
safe
ver-
bos
wit
rng
cal
bac

Bases: spynnaker.pyNN.models.neural_projections.connectors.fixed_number_post_connector.FixedNumberPostConnector, pyNN.connectors.FixedNumberPostConnector

PyNN connector that puts a fixed number of connections on each of the post 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*) – 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** – random number generator
- **callback** – list of callbacks to run

spynnaker8.models.connectors.fixed_number_pre_connector module

class spynnaker8.models.connectors.fixed_number_pre_connector.**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, pyNN.connectors.FixedNumberPreConnector

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 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** –
- **callback** –

spynnaker8.models.connectors.fixed_probability_connector module

class spynnaker8.models.connectors.fixed_probability_connector.**FixedProbabilityConnector**(*p_*
al
lo
sa
ve
bo
rn
ca
ba

Bases: spynnaker.pyNN.models.neural_projections.connectors.fixed_probability_connector.FixedProbabilityConnector, pyNN.connectors.FixedProbabilityConnector

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

Parameters

- **p_connect** (*float*) – a number between zero and one. Each potential connection is created with this probability.
- **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.
- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **space** – a Space object, needed if you wish to specify distance-dependent weights or delays - not implemented
- **verbose** –
- **rng** –
- **callback** –

p_connect

spynnaker8.models.connectors.from_file_connector module

class spynnaker8.models.connectors.from_file_connector.**FromFileConnector**(*file*, *distributed=False*, *safe=True*, *callback=None*, *verbose=False*)

Bases: *spynnaker8.models.connectors.from_list_connector.FromListConnector*, pyNN.connectors.FromFileConnector

get_reader(*file*)

Get a file reader object using the PyNN methods.

Returns A pynn StandardTextFile or similar

spynnaker8.models.connectors.from_list_connector module

class spynnaker8.models.connectors.from_list_connector.**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.

Parameters

- **conn_list** – 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).
- **column_names** – the names of the parameters p1, p2, etc. If not provided, it is assumed the parameters are weight, delay (for backwards compatibility).
- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** – if given, a callable that display a progress bar on the terminal.

spynnaker8.models.connectors.index_based_probability_connector module

class spynnaker8.models.connectors.index_based_probability_connector.**IndexBasedProbability**

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.

Parameters

- **index_expression** (*str*) – a function of the indices of the populations An expression
- **allow_self_connections** (*bool*) – allow a neuron to connect to itself

spynnaker8.models.connectors.kernel_connector module

class spynnaker8.models.connectors.kernel_connector.**KernelConnector**(*shape_pre*, *shape_post*, *shape_kernel*, *weight_kernel=None*, *delay_kernel=None*, *shape_common=None*, *pre_sample_steps=None*, *pre_start_coords=None*, *post_sample_steps=None*, *post_start_coords=None*, *safe=True*, *space=None*, *verbose=False*)

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.

TODO: should these include allow_self_connections and with_replacement?

Parameters

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

spynnaker8.models.connectors.multapse_connector module

class spynnaker8.models.connectors.multapse_connector.**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. Uniform selection probability is assumed.

Parameters

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

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

spynnaker8.models.connectors.one_to_one_connector module

class spynnaker8.models.connectors.one_to_one_connector.**OneToOneConnector**(*safe=True*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.one_to_one_connector.OneToOneConnector, pyNN.connectors.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.

Parameters

- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** – a function that will be called with the fractional progress of the connection routine. An example would be progress_bar.set_level.

spynnaker8.models.connectors.small_world_connector module

class spynnaker8.models.connectors.small_world_connector.**SmallWorldConnector**(*degree*, *rewiring*, *allow_self_connection*, *safe=True*, *verbose=False*, *n_connections=Non*

Bases: spynnaker.pyNN.models.neural_projections.connectors.small_world_connector.SmallWorldConnector

Module contents

class spynnaker8.models.connectors.**AllToAllConnector**(*allow_self_connections=True*, *safe=True*, *verbose=None*, *callbacks=None*)

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

all_to_all_connector.AllToAllConnector, pyNN.connectors.AllToAllConnector

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** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **verbose** –
- **callbacks** –

class spynnaker8.models.connectors.**ArrayConnector**(*array*, *safe=True*, *callback=None*, *verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.array_connector.ArrayConnector

Create an array connector.

Parameters **array** (*integer*) – an array of integers

class spynnaker8.models.connectors.**CSAConnector**(*cset*, *safe=True*, *callback=None*, *verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.csa_connector.CSAConnector

Create an CSA (Connection Set Algebra, Djurfeldt 2012) connector.

Parameters **cset** (*string*) – a connection set description

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, pyNN.connectors.DistanceDependentProbabilityConnector

Make connections using a distribution which varies with distance.

Parameters

- **d_expression** (*string*) – 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.
- **n_connections** (*int*) – The number of efferent synaptic connections per neuron.

- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.

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.connectors.FixedNumberPostConnector

PyNN connector that puts a fixed number of connections on each of the post 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*) – 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** – random number generator
- **callback** – list of callbacks to run

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, pyNN.connectors.FixedNumberPreConnector

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 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** –
- **callback** –

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, pyNN.connectors.FixedProbabilityConnector

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

Parameters

- **p_connect** (*float*) – a number between zero and one. Each potential connection is created with this probability.
- **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.
- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **space** – a Space object, needed if you wish to specify distance-dependent weights or delays - not implemented
- **verbose** –
- **rng** –
- **callback** –

p_connect

class spynnaker8.models.connectors.**FromFileConnector**(*file*, *distributed=False*, *safe=True*, *callback=None*, *verbose=False*)

Bases: *spynnaker8.models.connectors.from_list_connector.FromListConnector*, pyNN.connectors.FromFileConnector

get_reader(*file*)

Get a file reader object using the PyNN methods.

Returns A pynn StandardTextFile or similar

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.

Parameters

- **conn_list** – 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).
- **column_names** – the names of the parameters p1, p2, etc. If not provided, it is assumed the parameters are weight, delay (for backwards compatibility).
- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** – if given, a callable that display a progress bar on the terminal.

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.

Parameters

- **index_expression** (*str*) – a function of the indices of the populations An expression
- **allow_self_connections** (*bool*) – allow a neuron to connect to itself

spynnaker8.models.connectors.**FixedTotalNumberConnector**

alias of *spynnaker8.models.connectors.multapse_connector.MultapseConnector*

class spynnaker8.models.connectors.**OneToOneConnector**(*safe=True*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.one_to_one_connector.OneToOneConnector, pyNN.connectors.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.

Parameters

- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** – a function that will be called with the fractional progress of the connection routine. An example would be progress_bar.set_level.

class spynnaker8.models.connectors.**SmallWorldConnector**(*degree*, *rewiring*, *allow_self_connections=True*, *safe=True*, *verbose=False*, *n_connections=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.small_world_connector.SmallWorldConnector

class spynnaker8.models.connectors.**KernelConnector**(*shape_pre*, *shape_post*, *shape_kernel*, *weight_kernel=None*, *delay_kernel=None*, *shape_common=None*, *pre_sample_steps=None*, *pre_start_coords=None*, *post_sample_steps=None*, *post_start_coords=None*, *safe=True*, *space=None*, *verbose=False*)

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.

TODO: should these include allow_self_connections and with_replacement?

Parameters

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

spynnaker8.models.populations package

Submodules

spynnaker8.models.populations.assembly module

class spynnaker8.models.populations.assembly.**Assembly**(**populations*, ***kwargs*)

Bases: pyNN.common.populations.Assembly

Create an Assembly of Populations and/or PopulationViews.

spynnaker8.models.populations.idmixin module

class spynnaker8.models.populations.idmixin.**IDMixin**(*population*, *id*)

Bases: object

as_view()
Return a PopulationView containing just this cell.

celltype

get_initial_value(*variable*)
Get the initial value of a state variable of the cell.

get_parameters()
Return a dict of all cell parameters.

id

inject(*current_source*)
Inject current from a current source object into the cell.

is_standard_cell

local

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.

set_initial_value(*variable*, *value*)
Set the initial value of a state variable of the cell.

set_parameters(***parameters*)
Set cell parameters, given as a sequence of parameter=value arguments.

spynnaker8.models.populations.population module

class spynnaker8.models.populations.population.**Population**(*size*, *cellclass*, *cellparams=None*, *structure=None*, *initial_values=None*, *label=None*, *constraints=None*, *additional_parameters=None*)
Bases: spynnaker.pyNN.models.pynn_population_common.PyNNPopulationCommon, *spynnaker8.models.recorder.Recorder*, *spynnaker8.models.populations.population_base.PopulationBase*

PyNN 0.8/0.9 population object

all()
Iterator over cell IDs on all MPI nodes.

all_cells
An array containing the cell IDs of all neurons in the Population (all MPI nodes).

annotations
The annotations given by the end user

can_record(*variable*)
Determine whether *variable* can be recorded from this population.

celltype
Implements the PyNN expected celltype property

Returns The celltype this property has been set to

static create(*cellclass*, *cellparams=None*, *n=1*)
Pass through method to the constructor defined by PyNN. Create n cells all of the same type. Returns a Population object.

Parameters

- **cellclass** – see Population.__init__
- **cellparams** – see Population.__init__
- **n** – see Population.__init__(size...)

Returns A New 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.

find_units(*variable*)
Get the units of a variable

Parameters **variable** – The name of the variable

Returns The units of the variable

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*) – 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*) – annotations to put on the neo block

Return type neo.Block

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*) – 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 get_data
- **clear** (*bool*) – Whether recorded data will be deleted.
- **annotations** (*dict*) – annotations to put on the neo block

Return type neo.Block

get_initial_value(*variable*, *selector=None*)
See AbstractPopulationInitializable.get_initial_value

get_initial_values(*selector=None*)
See AbstractPopulationInitializable.get_initial_values

get_spike_counts(*gather=True*)
Return the number of spikes for each neuron.

initial_values

initialize(***kwargs*)

position_generator
NO PyNN description of this method.

positions
Return the position array for structured populations.

record(*variables*, *to_file=None*, *sampling_interval=None*, *indexes=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** (*a Neo IO instance*) – 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.
- **indexes** – The indexes of neurons to record from. This is none Standard PyNN and equivelent to creating a view with these indexes and asking the View to record.

sample(*n*, *rng=None*)
Randomly sample *n* cells from the Population, and return a PopulationView object.

set(***parameters*)
Set one or more parameters for every cell in the population.

param can be a dict, in which case value should not be supplied, or a string giving the parameter name, in which case value is the parameter value. value can be a numeric value, or list of such (e.g. for setting spike times):

```
p.set("tau_m", 20.0).
p.set({'tau_m':20, 'v_rest':-65})
```

Parameters

- **parameter** (*str or dict*) – the parameter to set
- **value** – the value of the parameter to set.

set_initial_value(*variable*, *value*, *selector=None*)
See AbstractPopulationInitializable.set_initial_value

spinnaker_get_data(*variable*)
Public accessor for getting data as a numpy array, instead of the neo based object

Parameters **variable** – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.

Returns numpy array of the data

tset(***kwargs*)
Deprecated. 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 instance 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** – pointless on sPyNNaker
- **clear** – clears the storage data if set to true after reading it back
- **annotations** – annotations to put on the neo block

spynnaker8.models.populations.population_base module

class spynnaker8.models.populations.population_base.**PopulationBase**
Bases: object

Shared methods between Populations and Population views.

Mainly pass through and not implemented

all_cells
An array containing the cell IDs of all neurons in the Population (all MPI nodes).

getSpikes(**args*, ***kwargs*)

> **Warning:** Deprecated. 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** – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** – 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.
- **clear** – If this is True, recorded data will be deleted from the Population.
- **annotations** – annotations to put on the neo block

get_gsyn (**args*, ***kwargs*)

> **Warning:** Deprecated. 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.

get_v (**args*, ***kwargs*)

> **Warning:** Deprecated. Use *get_data('v')* instead.

inject (*current_source*)
Connect a current source to all cells in the Population.

is_local (*id*)
Indicates whether the cell with the given ID exists on the local MPI node.

local_cells
An array containing the cell IDs of those neurons in the Population that exist on the local MPI node.

local_size
Return the number of cells in the population on the local MPI node.

meanSpikeCount (**args*, ***kwargs*)

> **Warning:** Deprecated. Use *mean_spike_count()* instead.

mean_spike_count (*gather=True*)
Returns the mean number of spikes per neuron.

nearest (*position*)
Return the neuron closest to the specified position.

position_generator

> **Warning:** NO PyNN description of this method.

positions

> **Warning:** NO PyNN description of this method.

printSpikes (*filename*, *gather=True*)

Warning: Deprecated. Use *write_data(file, 'spikes')* instead.

Note: Method signature is the PyNN0.7 one

print_gsyn(*filename*, *gather=True*)

Warning: Deprecated. Use *write_data(file, ['gsyn_exc', 'gsyn_inh'])* instead.

Note: Method signature is the PyNN0.7 one

print_v(*filename*, *gather=True*)

Warning: Deprecated. Use *write_data(file, 'v')* instead.

Note: Method signature is the PyNN0.7 one

receptor_types()
NO PyNN description of this method.

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** (*a Neo IO instance*) – 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*)

Warning: Deprecated. 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*)

> **Warning:** Deprecated. 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*)

> **Warning:** Deprecated. Use *set(parametername=rand_distr)* instead.

save_positions(*file*)
Save positions to file. The output format is index x y z

structure
The spatial structure of the parent Population.

tset(***kwargs*)

> **Warning:** Deprecated. 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 instance 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** – pointless on sPyNNaker
- **clear** – clears the storage data if set to true after reading it back
- **annotations** – annotations to put on the Neo block

spynnaker8.models.populations.population_view module

class spynnaker8.models.populations.population_view.**PopulationView**(*parent*, *selector*, *label=None*)

Bases: *spynnaker8.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 **selector** – 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.

all()
Iterator over cell IDs (on all MPI nodes).

all_cells
An array containing the cell IDs of all neurons in the Population (all MPI nodes).

can_record(*variable*)
Determine whether variable can be recorded from this population.

celltype
The type of neurons making up the Population.

conductance_based
Indicates whether the post-synaptic response is modelled as a change in conductance or a change in current.

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.

find_units(*variable*)

Warning: NO PyNN description of this method.

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).

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** – Either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** – 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** – If True, recorded data will be deleted from the Population. :param annotations: annotations to put on the neo block

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 tbe Populations uses PyNN 7 version of the get_spikes method which dooes not support indexes.

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.

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

index_in_grandparent(*indices*)
Given an array of indices, return the indices in the parent population at the root of the tree.

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

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, 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)
```

label
A label for the Population.

mask
The selector mask that was used to create this view.

parent
A reference to the parent Population (that this is a view of).

record(*variables*, *to_file=None*, *sampling_interval=None*)
Record the specified variable or variables for all cells in the Population or view.

Parameters

- **variables** – 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** – If specified, should be a Neo IO instance and write_data() will be automatically called when end() is called.
- **sampling_interval** – 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, and return a PopulationView object.

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.

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** – a Neo IO instance
- **variables** – either a single variable name or a list of variable names. These must have been previously recorded, otherwise an Exception will be raised.
- **gather** – 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.
- **clear** – If this is True, recorded data will be deleted from the Population.
- **annotations** – 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.

Module contents

class spynnaker8.models.populations.**Assembly**(**populations*, ***kwargs*)
Bases: pyNN.common.populations.Assembly

Create an Assembly of Populations and/or PopulationViews.

class spynnaker8.models.populations.**IDMixin**(*population*, *id*)
Bases: object

as_view()
Return a PopulationView containing just this cell.

celltype

get_initial_value(*variable*)
Get the initial value of a state variable of the cell.

get_parameters()
Return a dict of all cell parameters.

id

inject(*current_source*)
Inject current from a current source object into the cell.

is_standard_cell

local

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.

set_initial_value(*variable*, *value*)
Set the initial value of a state variable of the cell.

set_parameters(***parameters*)
Set cell parameters, given as a sequence of parameter=value arguments.

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.pynn_population_common.PyNNPopulationCommon, *spynnaker8.models.recorder.Recorder*, *spynnaker8.models.populations.population_base.PopulationBase*

PyNN 0.8/0.9 population object

all()
Iterator over cell IDs on all MPI nodes.

all_cells
An array containing the cell IDs of all neurons in the Population (all MPI nodes).

annotations
The annotations given by the end user

can_record(*variable*)
Determine whether *variable* can be recorded from this population.

celltype
Implements the PyNN expected celltype property

Returns The celltype this property has been set to

static create(*cellclass*, *cellparams=None*, *n=1*)
Pass through method to the constructor defined by PyNN. Create n cells all of the same type. Returns a Population object.

Parameters

- **cellclass** – see Population.__init__
- **cellparams** – see Population.__init__
- **n** – see Population.__init__(size...)

Returns A New 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.

find_units(*variable*)
Get the units of a variable

Parameters **variable** – The name of the variable

Returns The units of the variable

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*) – 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*) – annotations to put on the neo block

Return type neo.Block

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*) – 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 get_data
- **clear** (*bool*) – Whether recorded data will be deleted.
- **annotations** (*dict*) – annotations to put on the neo block

Return type neo.Block

get_initial_value(*variable*, *selector=None*)
See AbstractPopulationInitializable.get_initial_value

get_initial_values(*selector=None*)
See AbstractPopulationInitializable.get_initial_values

get_spike_counts(*gather=True*)
Return the number of spikes for each neuron.

initial_values

initialize(***kwargs*)

position_generator
NO PyNN description of this method.

positions
Return the position array for structured populations.

record(*variables*, *to_file=None*, *sampling_interval=None*, *indexes=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** (*a Neo IO instance*) – 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.
- **indexes** – The indexes of neurons to record from. This is none Standard PyNN and equivelent to creating a view with these indexes and asking the View to record.

sample(*n*, *rng=None*)
Randomly sample *n* cells from the Population, and return a PopulationView object.

set(***parameters*)
Set one or more parameters for every cell in the population.

param can be a dict, in which case value should not be supplied, or a string giving the parameter name, in which case value is the parameter value. value can be a numeric value, or list of such (e.g. for setting spike times):

```
p.set("tau_m", 20.0).
p.set({'tau_m':20, 'v_rest':-65})
```

Parameters

- **parameter** (*str or dict*) – the parameter to set
- **value** – the value of the parameter to set.

set_initial_value(*variable*, *value*, *selector=None*)
See AbstractPopulationInitializable.set_initial_value

spinnaker_get_data(*variable*)
Public accessor for getting data as a numpy array, instead of the neo based object

Parameters **variable** – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.

Returns numpy array of the data

tset(***kwargs*)
Deprecated. 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 instance 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** – pointless on sPyNNaker
- **clear** – clears the storage data if set to true after reading it back
- **annotations** – annotations to put on the neo block

class spynnaker8.models.populations.**PopulationBase**
Bases: object

Shared methods between Populations and Population views.

Mainly pass through and not implemented

all_cells
An array containing the cell IDs of all neurons in the Population (all MPI nodes).

getSpikes(**args*, ***kwargs*)

Warning: Deprecated. 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** – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** – 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.
- **clear** – If this is True, recorded data will be deleted from the Population.
- **annotations** – annotations to put on the neo block

get_gsyn(**args*, ***kwargs*)

Warning: Deprecated. 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.

get_v(**args*, ***kwargs*)

> **Warning:** Deprecated. Use *get_data('v')* instead.

inject(*current_source*)
Connect a current source to all cells in the Population.

is_local(*id*)
Indicates whether the cell with the given ID exists on the local MPI node.

local_cells
An array containing the cell IDs of those neurons in the Population that exist on the local MPI node.

local_size
Return the number of cells in the population on the local MPI node.

meanSpikeCount(**args*, ***kwargs*)

> **Warning:** Deprecated. Use *mean_spike_count()* instead.

mean_spike_count(*gather=True*)
Returns the mean number of spikes per neuron.

nearest(*position*)
Return the neuron closest to the specified position.

position_generator

> **Warning:** NO PyNN description of this method.

positions

> **Warning:** NO PyNN description of this method.

printSpikes(*filename*, *gather=True*)

> **Warning:** Deprecated. Use *write_data(file, 'spikes')* instead.

> **Note:** Method signature is the PyNN0.7 one

print_gsyn(*filename*, *gather=True*)

Warning: Deprecated. Use *write_data(file, ['gsyn_exc', 'gsyn_inh'])* instead.

Note: Method signature is the PyNN0.7 one

print_v(*filename*, *gather=True*)

Warning: Deprecated. Use *write_data(file, 'v')* instead.

Note: Method signature is the PyNN0.7 one

receptor_types()
NO PyNN description of this method.

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** (*a Neo IO instance*) – 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*)

Warning: Deprecated. 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*)

Warning: Deprecated. 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*)

> **Warning:** Deprecated. Use *set(parametername=rand_distr)* instead.

save_positions(*file*)
Save positions to file. The output format is index x y z

structure
The spatial structure of the parent Population.

tset(***kwargs*)

> **Warning:** Deprecated. 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 instance 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** – pointless on sPyNNaker
- **clear** – clears the storage data if set to true after reading it back
- **annotations** – annotations to put on the Neo block

class spynnaker8.models.populations.**PopulationView**(*parent*, *selector*, *label=None*)
Bases: *spynnaker8.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 **selector** – 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.

all()
Iterator over cell IDs (on all MPI nodes).

all_cells
An array containing the cell IDs of all neurons in the Population (all MPI nodes).

can_record(*variable*)
Determine whether variable can be recorded from this population.

celltype
The type of neurons making up the Population.

conductance_based
Indicates whether the post-synaptic response is modelled as a change in conductance or a change in current.

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.

find_units(*variable*)

> **Warning:** NO PyNN description of this method.

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).

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** – Either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** – 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** – If True, recorded data will be deleted from the Population. :param annotations: annotations to put on the neo block

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 tbe Populations uses PyNN 7 version of the get_spikes method which dooes not support indexes.

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.

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

index_in_grandparent(*indices*)
Given an array of indices, return the indices in the parent population at the root of the tree.

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

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, 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)
```

label
A label for the Population.

mask
The selector mask that was used to create this view.

parent
A reference to the parent Population (that this is a view of).

record(*variables*, *to_file=None*, *sampling_interval=None*)
Record the specified variable or variables for all cells in the Population or view.

Parameters

- **varables** – 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** – If specified, should be a Neo IO instance and write_data() will be automatically called when end() is called.
- **sampling_interval** – 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, and return a PopulationView object.

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.

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** – a Neo IO instance
- **variables** – either a single variable name or a list of variable names. These must have been previously recorded, otherwise an Exception will be raised.
- **gather** – 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.
- **clear** – If this is True, recorded data will be deleted from the Population.
- **annotations** – 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.

spynnaker8.models.synapse_dynamics package

Subpackages

spynnaker8.models.synapse_dynamics.timing_dependence package

Submodules

spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_pfister_spike_triplet module

class spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_pfister_spike_

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_pfister_spike_triplet.TimingDependencePfisterSpikeTriplet

A_minus

A_plus

spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_recurrent module

class spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_recurrent.**Tim**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_recurrent.TimingDependenceRecurrent

A_minus

A_plus

spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_spike_nearest_pair module

class spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_spike_nearest_

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spike_nearest_pair.TimingDependenceSpikeNearestPair

A_minus

A_plus

spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_spike_pair module

class spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_spike_pair.**Tir**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spike_pair.TimingDependenceSpikePair

A_minus

A_plus

spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_vogels_2011 module

class spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_vogels_2011.**T**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_vogels_2011.TimingDependenceVogels2011

A_minus

A_plus

Module contents

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

A_minus

A_plus

class spynnaker8.models.synapse_dynamics.timing_dependence.**TimingDependencePfisterSpikeTrip**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_pfister_spike_triplet.TimingDependencePfisterSpikeTriplet

A_minus

A_plus

class spynnaker8.models.synapse_dynamics.timing_dependence.**TimingDependenceRecurrent**(*accumu*, *6*, *accumulator_pote*, *mean_p*, *mean_p*, *dual_fsr*, *A_plus=*, *A_minu*

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_recurrent.TimingDependenceRecurrent

A_minus

A_plus

class spynnaker8.models.synapse_dynamics.timing_dependence.**TimingDependenceSpikeNearestPair**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_spike_nearest_pair.TimingDependenceSpikeNearestPair

A_minus

A_plus

class spynnaker8.models.synapse_dynamics.timing_dependence.**TimingDependenceVogels2011**(*alpha*, *tau=20*, *A_plus*, *A_min*

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.timing_dependence.timing_dependence_vogels_2011.TimingDependenceVogels2011

A_minus

A_plus

spynnaker8.models.synapse_dynamics.weight_dependence package

Submodules

spynnaker8.models.synapse_dynamics.weight_dependence.weight_dependence_additive module

class spynnaker8.models.synapse_dynamics.weight_dependence.weight_dependence_additive.**Weigh**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_additive.WeightDependenceAdditive

spynnaker8.models.synapse_dynamics.weight_dependence.weight_dependence_additive_triplet module

class spynnaker8.models.synapse_dynamics.weight_dependence.weight_dependence_additive_tripl

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_additive_triplet.WeightDependenceAdditiveTriplet

spynnaker8.models.synapse_dynamics.weight_dependence.weight_dependence_multiplicative module

class spynnaker8.models.synapse_dynamics.weight_dependence.weight_dependence_multiplicative

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_multiplicative.WeightDependenceMultiplicative

Module contents

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

class spynnaker8.models.synapse_dynamics.weight_dependence.**WeightDependenceMultiplicative**(*w*
w

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_multiplicative.WeightDependenceMultiplicative

class spynnaker8.models.synapse_dynamics.weight_dependence.**WeightDependenceAdditiveTriplet**

Bases: spynnaker.pyNN.models.neuron.plasticity.stdp.weight_dependence.weight_dependence_additive_triplet.WeightDependenceAdditiveTriplet

Submodules

spynnaker8.models.synapse_dynamics.synapse_dynamics_static module

class spynnaker8.models.synapse_dynamics.synapse_dynamics_static.**SynapseDynamicsStatic**(*weig*
de-
lay=

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_static.SynapseDynamicsStatic

spynnaker8.models.synapse_dynamics.synapse_dynamics_stdp module

class spynnaker8.models.synapse_dynamics.synapse_dynamics_stdp.**SynapseDynamicsSTDP**(*timing_dep*
weight_dep
volt-
age_depen
den-
dritic_dela
weight=0.0
de-
lay=None,
back-
prop_delay

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_stdp.SynapseDynamicsSTDP

spynnaker8.models.synapse_dynamics.synapse_dynamics_structural_static module

class spynnaker8.models.synapse_dynamics.synapse_dynamics_structural_static.**SynapseDynamics**

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_structural_static.SynapseDynamicsStructuralStatic

spynnaker8.models.synapse_dynamics.synapse_dynamics_structural_stdp module

class spynnaker8.models.synapse_dynamics.synapse_dynamics_structural_stdp.**SynapseDynamicsSt**

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.

synapse_dynamics_structural_stdp.SynapseDynamicsStructuralSTDP

Module contents

class spynnaker8.models.synapse_dynamics.**SynapseDynamicsStatic**(*weight=0.0*, *delay=None*)

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_static.SynapseDynamicsStatic

class spynnaker8.models.synapse_dynamics.**SynapseDynamicsSTDP**(*timing_dependence*, *weight_dependence*, *voltage_dependence=None*, *dendritic_delay_fraction=1.0*, *weight=0.0*, *delay=None*, *backprop_delay=True*)

Bases: spynnaker.pyNN.models.neuron.synapse_dynamics.synapse_dynamics_stdp.SynapseDynamicsSTDP

1.1.3.2 Submodules

1.1.3.3 spynnaker8.models.data_cache module

class spynnaker8.models.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** – cache label
- **description** – cache description
- **segment_number** – cache segment number
- **recording_start_time** – when this cache was started in recording space.
- **t** – time

description

get_data(*variable*)

Get the variable cache for the named variable

Parameters **variable** – name of variable to get cache for

Rtype variable str

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** (*nparray*) – raw data in sPyNNaker format
- **indexes** (*nparray*) – 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

Return type None

segment_number

t

variables
Provides a list of which variables data has been cached for

Return type Iterator (str)

1.1.3.4 spynnaker8.models.projection module

class spynnaker8.models.projection.**Projection**(*pre_synaptic_population*, *post_synaptic_population*, *connector*, *synapse_type=None*, *source=None*, *receptor_type=None*, *space=None*, *label=None*)
Bases: spynnaker.pyNN.models.pynn_projection_common.PyNNProjectionCommon

sPyNNaker 8 projection class

get(*attribute_names*, *format*, *gather=True*, *with_address=True*, *multiple_synapses='last'*)
Get a parameter for PyNN 0.8

Parameters

- **attribute_names** (*str or iterable(str)*) – list of attributes to gather
- **format** – “list” or “array”
- **gather** – gather over all nodes (defaulted to true on SpiNNaker)
- **with_address** – True if the source and target are to be included

- **multiple_synapses** – 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*)

getSynapseDynamics (*parameter_name*, *format='list'*, *gather=True*)

getWeights (*format='list'*, *gather=True*)

label

post

pre

printDelays (*file*, *format='list'*, *gather=True*)
Print synaptic weights to file. In the array format, zeros are printed for non-existent connections.

printWeights (*file*, *format='list'*, *gather=True*)

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).

saveConnections (*file*, *gather=True*, *compatible_output=True*)

set (***attributes*)

weightHistogram (*min=None*, *max=None*, *nbins=10*)
Return a histogram of synaptic weights. If min and max are not given, the minimum and maximum weights are calculated automatically.

1.1.3.5 spynnaker8.models.recorder module

class spynnaker8.models.recorder.**Recorder** (*population*)
Bases: spynnaker.pyNN.models.recording_common.RecordingCommon

cache_data ()
Store data for later extraction

read_in_signal (*segment*, *block*, *signal_array*, *data_indexes*, *view_indexes*, *variable*, *recording_start_time*, *sampling_interval*, *units*, *label*)
Reads in a data item that's not spikes (likely v, gsyn e, gsyn i) and saves this data to the segment.

Parameters

- **segment** (*neo.Segment*) – Segment to add data to
- **block** (*neo.Block*) – neo block
- **signal_array** (*nparray*) – the raw signal data
- **data_indexes** (*list(int)*) – The indexes for the recorded data
- **view_indexes** (*list(int)*) – The indexes for which data should be returned. If None all data (view_index = data_indexes)
- **variable** – the variable name
- **recording_start_time** – when recording started
- **sampling_interval** – how often a neuron is recorded

- **units** – the units of the recorded value
- **label** – human readable label

read_in_spikes(*segment*, *spikes*, *t*, *n_neurons*, *recording_start_time*, *sampling_interval*, *indexes*, *label*)
Converts the data into SpikeTrains and saves them to the segment.

Parameters

- **segment** (*neo.Segment*) – Segment to add spikes to
- **spikes** (*nparray*) – Spike data in raw sPyNNaker format
- **t** (*int*) – last simulation time
- **n_neurons** (*int*) – total number of neurons including ones not recording
- **recording_start_time** (*int*) – time recording started
- **sampling_interval** – how often a neuron is recorded
- **label** (*str*) – recording elements label

1.1.3.6 spynnaker8.models.variable_cache module

class spynnaker8.models.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** (*nparray*) – 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

data

indexes

n_neurons

sampling_interval

units

1.1.3.7 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.pynn_projection_common.PyNNProjectionCommon

sPyNNaker 8 projection class

get(*attribute_names*, *format*, *gather=True*, *with_address=True*, *multiple_synapses='last'*)
Get a parameter for PyNN 0.8

Parameters

- **attribute_names** (*str or iterable(str)*) – list of attributes to gather
- **format** – "list" or "array"
- **gather** – gather over all nodes (defaulted to true on SpiNNaker)
- **with_address** – True if the source and target are to be included
- **multiple_synapses** – 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*)

getSynapseDynamics(*parameter_name*, *format='list'*, *gather=True*)

getWeights(*format='list'*, *gather=True*)

label

post

pre

printDelays(*file*, *format='list'*, *gather=True*)
Print synaptic weights to file. In the array format, zeros are printed for non-existent connections.

printWeights(*file*, *format='list'*, *gather=True*)

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).

saveConnections(*file*, *gather=True*, *compatible_output=True*)

set(***attributes*)

weightHistogram(*min=None*, *max=None*, *nbins=10*)
Return a histogram of synaptic weights. If min and max are not given, the minimum and maximum weights are calculated automatically.

class spynnaker8.models.**Recorder**(*population*)
Bases: spynnaker.pyNN.models.recording_common.RecordingCommon

cache_data()
Store data for later extraction

read_in_signal(*segment*, *block*, *signal_array*, *data_indexes*, *view_indexes*, *variable*, *recording_start_time*, *sampling_interval*, *units*, *label*)
Reads in a data item that's not spikes (likely v, gsyn e, gsyn i) and saves this data to the segment.

Parameters

- **segment** (*neo.Segment*) – Segment to add data to
- **block** (*neo.Block*) – neo block
- **signal_array** (*nparray*) – the raw signal data
- **data_indexes** (*list(int)*) – The indexes for the recorded data

- **view_indexes** (*list(int)*) – The indexes for which data should be returned. If None all data (view_index = data_indexes)
- **variable** – the variable name
- **recording_start_time** – when recording started
- **sampling_interval** – how often a neuron is recorded
- **units** – the units of the recorded value
- **label** – human readable label

read_in_spikes(*segment*, *spikes*, *t*, *n_neurons*, *recording_start_time*, *sampling_interval*, *indexes*, *label*)

Converts the data into SpikeTrains and saves them to the segment.

Parameters

- **segment** (*neo.Segment*) – Segment to add spikes to
- **spikes** (*nparray*) – Spike data in raw sPyNNaker format
- **t** (*int*) – last simulation time
- **n_neurons** (*int*) – total number of neurons including ones not recording
- **recording_start_time** (*int*) – time recording started
- **sampling_interval** – how often a neuron is recorded
- **label** (*str*) – recording elements label

1.1.4 spynnaker8.utilities package

1.1.4.1 Subpackages

spynnaker8.utilities.random_stats package

Submodules

spynnaker8.utilities.random_stats.random_stats_binomial_impl module

class spynnaker8.utilities.random_stats.random_stats_binomial_impl.**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

spynnaker8.utilities.random_stats.random_stats_exponential_impl module

class spynnaker8.utilities.random_stats.random_stats_exponential_impl.**RandomStatsExponentia**
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

spynnaker8.utilities.random_stats.random_stats_gamma_impl module

class spynnaker8.utilities.random_stats.random_stats_gamma_impl.**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

spynnaker8.utilities.random_stats.random_stats_log_normal_impl module

class spynnaker8.utilities.random_stats.random_stats_log_normal_impl.**RandomStatsLogNormalIm**
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

spynnaker8.utilities.random_stats.random_stats_normal_clipped_impl module

class spynnaker8.utilities.random_stats.random_stats_normal_clipped_impl.**RandomStatsNormalC**
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

spynnaker8.utilities.random_stats.random_stats_normal_impl module

class spynnaker8.utilities.random_stats.random_stats_normal_impl.**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

spynnaker8.utilities.random_stats.random_stats_poisson_impl module

class spynnaker8.utilities.random_stats.random_stats_poisson_impl.**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

spynnaker8.utilities.random_stats.random_stats_randint_impl module

class spynnaker8.utilities.random_stats.random_stats_randint_impl.**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

spynnaker8.utilities.random_stats.random_stats_scipy_impl module

class spynnaker8.utilities.random_stats.random_stats_scipy_impl.**RandomStatsScipyImpl**(*distribu*
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

spynnaker8.utilities.random_stats.random_stats_uniform_impl module

class spynnaker8.utilities.random_stats.random_stats_uniform_impl.**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

spynnaker8.utilities.random_stats.random_stats_vonmises_impl module

class spynnaker8.utilities.random_stats.random_stats_vonmises_impl.**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

Module contents

class spynnaker8.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 spynnaker8.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 spynnaker8.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 spynnaker8.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 spynnaker8.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 spynnaker8.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 spynnaker8.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 spynnaker8.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 spynnaker8.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 spynnaker8.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 spynnaker8.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

1.1.4.2 Submodules

1.1.4.3 spynnaker8.utilities.exceptions module

exception spynnaker8.utilities.exceptions.**DelayExtensionException**
Bases: spinn_front_end_common.utilities.exceptions.ConfigurationException

Raised when a delay extension vertex fails

exception spynnaker8.utilities.exceptions.**FilterableException**
Bases: *spynnaker8.utilities.exceptions.Spynnaker8Exception*

Raised when it is not possible to determine if an edge should be filtered

exception spynnaker8.utilities.exceptions.**InvalidParameterType**
Bases: *spynnaker8.utilities.exceptions.Spynnaker8Exception*

Raised when a parameter is not recognised

exception spynnaker8.utilities.exceptions.**MemReadException**
Bases: *spynnaker8.utilities.exceptions.Spynnaker8Exception*

Raised when the pynn front end fails to read a certain memory region

exception spynnaker8.utilities.exceptions.**Spynnaker8Exception**
Bases: Exception

Superclass of all exceptions from the pynn module

exception spynnaker8.utilities.exceptions.**SynapticBlockGenerationException**
Bases: spinn_front_end_common.utilities.exceptions.ConfigurationException

Raised when the synaptic manager fails to generate a synaptic block

exception spynnaker8.utilities.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 spynnaker8.utilities.exceptions.**SynapticConfigurationException**
Bases: spinn_front_end_common.utilities.exceptions.ConfigurationException

Raised when the synaptic manager fails for some reason

exception spynnaker8.utilities.exceptions.**SynapticMaxIncomingAtomsSupportException**
Bases: spinn_front_end_common.utilities.exceptions.ConfigurationException

Raised when a synaptic sublist exceeds the max atoms possible to be supported

1.1.4.4 spynnaker8.utilities.id module

class spynnaker8.utilities.id.**ID**(*n*)
Bases: int, pyNN.common.populations.IDMixin

A filter container for allowing random setters of values

Create an ID object with numerical value *n*.

1.1.4.5 spynnaker8.utilities.neo_compare module

spynnaker8.utilities.neo_compare.**compare_analogsignal**(*as1*, *as2*, *same_length=True*)
Compares two analogsignalarray 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 analogsignalarrays are not equal

spynnaker8.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** – 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.

Return type None

Raises **AssertionError** – If the blocks are not equal

spynnaker8.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.

Return type None

Raises **AssertionError** – If the segments are not equal

spynnaker8.utilities.neo_compare.**compare_spiketrain**(*spiketrain1*, *spiketrain2*, *same_length=True*)

Checks two Spiketrains have the exact same data

Parameters

- **spiketrain1** (*SpikeTrain*) – first spiketrain
- **spiketrain2** – 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

spynnaker8.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

1.1.4.6 spynnaker8.utilities.neo_convertor module

spynnaker8.utilities.neo_convertor.**convert_analog_signal**(*signal_array*, *time_unit=UnitTime('millisecond', 0.001 * s, 'ms')*)

Converts part of a NEO object into told spynakker7 format

Parameters

- **signal_array** – Extended Quantities object
- **time_unit** – 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

Parameters

- **data** (*SpynnakerNeoBlock*) – 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 nparray

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

Parameters

- **data** (*SpynnakerNeoBlock*) – Data as returned by a getData() call
- **name** (*str*) – Name of the data to be extracted. Same values as used in getData()
- **runs** – List of Zero based index of the run to extract data for. Or None to extract all runs

Return type list(nparray)

spynnaker8.utilities.neo_convertor.**convert_gsyn**(*gsyn_exc*, *gsyn_inh*)

Converts two neo objects into the spynakker7 format

Note: It is acceptable for both neo parameters to be the same object

Parameters

- **gsyn_exc** – neo with gsyn_exc data
- **gsyn_inh** – neo with gsyn_exc data

Return type nparray

spynnaker8.utilities.neo_convertor.**convert_gsyn_exc_list**(*data*)
Converts the gsyn_exc into a list numpy array one per segment (all runs) in the format ID, time, value

Parameters **data** (*SpynnakerNeoBlock*) – The data to convert; it must have Gsyn_exc data in it

Return type list(nparray)

spynnaker8.utilities.neo_convertor.**convert_gsyn_inh_list**(*data*)
Converts the gsyn_inh into a list numpy array one per segment (all runs) in the format ID, time, value

Parameters **data** (*SpynnakerNeoBlock*) – The data to convert; it must have Gsyn_inh data in it

Return type list(nparray)

spynnaker8.utilities.neo_convertor.**convert_spikes**(*neo*, *run=0*)
Extracts the spikes for run one from a Neo Object

Parameters

- **neo** – neo Object including Spike Data
- **run** (*int*) – Zero based index of the run to extract data for

Return type nparray

spynnaker8.utilities.neo_convertor.**convert_spiketrains**(*spiketrains*)
Converts a list of spiketrains into spynakker7 format

Parameters **spiketrains** – List of SpikeTrains

Return type nparray

spynnaker8.utilities.neo_convertor.**convert_v_list**(*data*)
Converts the voltage into a list numpy array one per segment (all runs) in the format ID, time, value

Parameters **data** (*SpynnakerNeoBlock*) – The data to convert; it must have V data in it

Return type list(nparray)

spynnaker8.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** – 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

Parameters **spiketrains** – List of SpikeTrains

Returns Total number of spikes in all the spiketrains

1.1.4.7 spynnaker8.utilities.version_util module

spynnaker8.utilities.version_util.**detect_supported_configuration**(*pynn_version*, *neo_version*)
Check if the version configuration of PyNN and Neo is one we support.

Note: We strongly encourage the use of PyNN 0.9 and Neo 0.6.

Returns True if we're using old PyNN 0.8 syntax

Raises **ImportError** – If a truly unsupported system is present or if we cannot parse the version numbers (shouldn't happen)

1.1.4.8 Module contents

class spynnaker8.utilities.**ID**(*n*)
Bases: int, pyNN.common.populations.IDMixin

A filter container for allowing random setters of values

Create an ID object with numerical value *n*.

1.2 Submodules

1.3 spynnaker8.setup_pynn module

spynnaker8.setup_pynn.**install_sPyNNaker8_into**(*module*)
Do the actual installation by creating a package within the given module's implementation. This is very nasty!

spynnaker8.setup_pynn.**version_satisfies**(*module*, *requirement*)
Perform a version check. This code could be smarter...

1.4 spynnaker8.spinnaker module

class spynnaker8.spinnaker.**SpiNNaker**(*database_socket_addresses*, *extra_algorithm_xml_paths*, *extra_mapping_inputs*, *extra_mapping_algorithms*, *extra_pre_run_algorithms*, *extra_post_run_algorithms*, *extra_load_algorithms*, *time_scale_factor*, *min_delay*, *max_delay*, *graph_label*, *n_chips_required=None*, *n_boards_required=None*, *timestep=0.1*, *hostname=None*)
Bases: spynnaker.pyNN.abstract_spinnaker_common.AbstractSpiNNakerCommon, pyNN.common.control.BaseState, *spynnaker8.spynnaker8_simulator_interface.Spynnaker8SimulatorInterface*

Main interface for the sPyNNaker implementation of PyNN 0.8/0.9

clear()
Clear the current recordings and reset the simulation

dt
The machine time step.

Returns the machine time step

get_distribution_to_stats()

get_pynn_NumpyRNG()

get_random_distribution()

is_a_pynn_random(*thing*)

mpi_rank
Gets the MPI rank of the simulator

Note: Meaningless on SpiNNaker, so we pretend we're the head node.

Returns Constant: 0

name
The name of the simulator. Used to ensure PyNN recording neo blocks are correctly labelled.

Returns the name of the simulator.

num_processes
Gets the number of MPI worker processes

Note: Meaningless on SpiNNaker, so we pretend there's one MPI process

Returns Constant: 1

populations
The list of all populations in the simulation.

Returns list of populations

projections
The list of all projections in the simulation.

Returns list of projections

recorders
The recorders, used by the PyNN state object

Returns the internal recorders object

reset()
Reset the state of the current network to time t = 0.

run(*run_time*)
Run the simulation for a span of simulation time.

Parameters **run_time** – the time to run for, in milliseconds

Returns None

run_until(*tstop*)
Run the simulation until the given simulation time.

Parameters **tstop** – when to run until in milliseconds

running
Whether the simulation is running or has run.

Note: Ties into our has_ran parameter for automatic pause and resume.

Returns the has_ran variable from the SpiNNaker main interface

segment_counter
The number of the current recording segment being generated.

Returns the segment counter

state
Used to bypass the dual level object

Returns the SpiNNaker object

Return type *spynnaker8.spinnaker.SpiNNaker*

t
The current simulation time

Returns the current runtime already executed

class spynnaker8.spinnaker.**Spynnaker8FailedState**
Bases: spynnaker.pyNN.utilities.spynnaker_failed_state.SpynnakerFailedState, *spynnaker8.spynnaker8_simulator_interface.Spynnaker8SimulatorInterface*

dt

mpi_rank

name

num_processes

recorders

segment_counter

t

write_on_end

1.5 spynnaker8.spynnaker8_simulator_interface module

class spynnaker8.spynnaker8_simulator_interface.**Spynnaker8SimulatorInterface**
Bases: spynnaker.pyNN.spynnaker_simulator_interface.SpynnakerSimulatorInterface

dt

mpi_rank

name

num_processes

recorders

segment_counter

t

1.6 spynnaker8.spynnaker_plotting module

Plotting tools to be used together with https://github.com/NeuralEnsemble/PyNN/blob/master/pyNN/utility/plotting.py

class spynnaker8.spynnaker_plotting.**SpynnakerPanel**(**data*, ***options*)
Bases: object

Represents a single panel in a multi-panel figure.

Compatible with pyNN.utility.plotting's Frame and can be mixed with pyNN.utility.plotting's Panel

Unlike pyNN.utility.plotting.Panel, Spikes are plotted faster, other data is plotted as a heatmap

A panel is a Matplotlib Axes or Subplot instance. A data item may be an AnalogSignalArray, or a list of SpikeTrains. The Panel will automatically choose an appropriate representation. Multiple data items may be plotted in the same panel.

Valid options are any valid Matplotlib formatting options that should be applied to the Axes/Subplot, plus in addition:

***data_labels*:** a list of strings of the same length as the number of data items.

***line_properties*:** a list of dicts containing Matplotlib formatting options, of the same length as the number of data items.

Whole Neo Objects can be passed in as long as they contain a single Segment/run and only contain one type of data Whole Segments can be passed in only if they only contain one type of data

plot(*axes*)
Plot the Panel's data in the provided Axes/Subplot instance.

spynnaker8.spynnaker_plotting.**handle_options**(*ax*, *options*)
Handles options that can not be passed to axes.plot

Removes the ones it has handled

axes.plot will throw an exception if it gets unwanted options

Parameters

- **ax** (*matplotlib.axes*) – An Axes in a matplot lib figure
- **options** – All options the plotter can be configured with

spynnaker8.spynnaker_plotting.**heat_plot**(*ax*, *neurons*, *times*, *values*, *label=''*, ***options*)
Plots three lists of neurons, times and values into a heatmap

Parameters

- **ax** – An Axes in a matplotlib figure
- **neurons** – List of neuron IDs
- **times** – List of times
- **values** – List of values to plot
- **label** – Label for the graph
- **options** – plotting options

spynnaker8.spynnaker_plotting.**heat_plot_neo**(*ax*, *signal_array*, *label=''*, ***options*)
Plots neurons, times and values into a heatmap

Parameters

- **ax** – An Axes in a matplot lib figure
- **signal_array** – Neo Signal array Object
- **label** – 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

Parameters

- **ax** – An Axes in a matplot lib figure
- **data** – nparray of values in spknakker7 format
- **label** – 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

If there is more than ode type of Data in the segment options must include the name of the data to plot

---

Note: method signature defined by pynn plotting. This allows mixing of this plotting tool and pynn's

---

Parameters

- **axes** – An Axes in a matplot lib figure
- **segment** – Data for one run to plot
- **label** – Label for the graph
- **options** – plotting options

spynnaker8.spynnaker_plotting.**plot_spikes**(*ax*, *spike_times*, *neurons*, *label=''*, ***options*)
Plots the spikes based on two lists

Parameters

- **ax** – An Axes in a matplot lib figure
- **spike_times** – List of Spiketimes
- **neurons** – List of Neuron Ids
- **label** – Label for the graph
- **options** – plotting options

spynnaker8.spynnaker_plotting.**plot_spikes_numpy**(*ax*, *spikes*, *label=''*, ***options*)
Plot all spikes

Parameters

- **ax** – An Axes in a matplot lib figure
- **spikes** – spynakker7 format nparray of spikes
- **label** – 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.

Parameters

- **ax** – An Axes in a matplot lib figure
- **spiketrains** – List of spiketimes
- **label** – Label for the graph
- **options** – plotting options

1.7 Module contents

class spynnaker8.**Cuboid**(*width*, *height*, *depth*)
Bases: pyNN.space.Shape

Represents a cuboidal volume within which neurons may be distributed.

Arguments:

height: extent in y direction

width: extent in x direction

depth: extent in z direction

sample(*n*, *rng*)
Return *n* points distributed randomly with uniform density within the cuboid.

spynnaker8.**distance**(*src*, *tgt*, *mask=None*, *scale_factor=1.0*, *offset=0.0*, *periodic_boundaries=None*)
Return the Euclidian distance between two cells.

Parameters

- **mask** – 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** – allows for different units in the pre- and post- position (the post-synaptic position is multiplied by this quantity).

class spynnaker8.**Grid2D**(*aspect_ratio=1.0*, *dx=1.0*, *dy=1.0*, *x0=0.0*, *y0=0.0*, *z=0*, *fill_order='sequential'*, *rng=None*)
Bases: pyNN.space.BaseStructure

Represents a structure with neurons distributed on a 2D grid.

Arguments:

***dx*, *dy*:** distances between points in the x, y directions.

***x0*, *y0*:** coordinates of the starting corner of the grid.

***z*:** the z-coordinate of all points in the grid.

***aspect_ratio*:** ratio of the number of grid points per side (not the ratio of the side lengths, unless `dx == dy`)

***fill_order*:** may be 'sequential' or 'random'

calculate_size(*n*)
docstring goes here

generate_positions(*n*)
Calculate and return the positions of *n* neurons positioned according to this structure.

parameter_names = ('aspect_ratio', 'dx', 'dy', 'x0', 'y0', 'z', 'fill_order')

class spynnaker8.**Grid3D**(*aspect_ratioXY=1.0*, *aspect_ratioXZ=1.0*, *dx=1.0*, *dy=1.0*, *dz=1.0*, *x0=0.0*, *y0=0.0*, *z0=0*, *fill_order='sequential'*, *rng=None*)
Bases: pyNN.space.BaseStructure

Represents a structure with neurons distributed on a 3D grid.

Arguments:

***dx*, *dy*, *dz*:** distances between points in the x, y, z directions.

***x0*, *y0*. *z0*:** coordinates of the starting corner of the grid.

***aspect_ratioXY*, *aspect_ratioXZ*:** ratios of the number of grid points per side (not the ratio of the side lengths, unless `dx == dy == dz`)

***fill_order*:** may be 'sequential' or 'random'.

If *fill_order* is 'sequential', the z-index will be filled first, then y then x, i.e. the first cell will be at (0,0,0) (given default values for the other arguments), the second at (0,0,1), etc.

calculate_size(*n*)
docstring goes here

generate_positions(*n*)
Calculate and return the positions of *n* neurons positioned according to this structure.

parameter_names = ('aspect_ratios', 'dx', 'dy', 'dz', 'x0', 'y0', 'z0', 'fill_order')

class spynnaker8.**Line**(*dx=1.0*, *x0=0.0*, *y=0.0*, *z=0.0*)
Bases: pyNN.space.BaseStructure

Represents a structure with neurons distributed evenly on a straight line.

Arguments:

***dx*:** distance between points in the line.

***y*, *z*,:** y- and z-coordinates of all points in the line.

***x0*:** x-coordinate of the first point in the line.

generate_positions(*n*)
Calculate and return the positions of *n* neurons positioned according to this structure.

parameter_names = ('dx', 'x0', 'y', 'z')

class spynnaker8.**NumpyRNG**(*seed=None*, *parallel_safe=True*)
Bases: pyNN.random.WrappedRNG

Wrapper for the numpy.random.RandomState class (Mersenne Twister PRNG).

normal_clipped(*mu=0.0*, *sigma=1.0*, *low=-inf*, *high=inf*, *size=None*)

normal_clipped_to_boundary(*mu=0.0*, *sigma=1.0*, *low=-inf*, *high=inf*, *size=None*)

translations = {'binomial': ('binomial', {'n': 'n', 'p': 'p'}), 'exponential': ('e

class spynnaker8.**RandomDistribution**(*distribution*, *parameters_pos=None*, *rng=None*, ***parameters_named*)
Bases: pyNN.random.RandomDistribution

Class which defines a next(n) method which returns an array of n random numbers from a given distribution.

Parameters

- **distribution** – the name of a random number distribution.
- **parameters_pos** – parameters of the distribution, provided as a tuple. For the correct ordering, see *random.available_distributions*.
- **rng** – the random number generator to use, if a specific one is desired (e.g., to provide a seed). If present, should be a *NumpyRNG*, GSLRNG or NativeRNG object.
- **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	

Create a new RandomDistribution.

class spynnaker8.**RandomStructure**(*boundary*, *origin=(0.0, 0.0, 0.0)*, *rng=None*)
Bases: pyNN.space.BaseStructure

Represents a structure with neurons distributed randomly within a given volume.

Arguments: *boundary* - a subclass of Shape. *origin* - the coordinates (x,y,z) of the centre of the volume.

generate_positions(*n*)
Calculate and return the positions of *n* neurons positioned according to this structure.

parameter_names = ('boundary', 'origin', 'rng')

class spynnaker8.**Space**(*axes=None*, *scale_factor=1.0*, *offset=0.0*, *periodic_boundaries=None*)
Bases: object

Class representing a space within distances can be calculated. The space is Cartesian, may be 1-, 2- or 3-dimensional, and may have periodic boundaries in any of the dimensions.

Arguments:

axes: if not supplied, then the 3D distance is calculated. If supplied, axes should be a string containing the axes to be used, e.g. 'x', or 'yz'. axes='xyz' is the same as axes=None.

scale_factor: it may be that the pre and post populations use different units for position, e.g. degrees and μm. In this case, *scale_factor* can be specified, which is applied to the positions in the post-synaptic population.

offset: if the origins of the coordinate systems of the pre- and post- synaptic populations are different, *offset* can be used to adjust for this difference. The offset is applied before any scaling.

periodic_boundaries: either *None*, or a tuple giving the boundaries for each dimension, e.g. *((x_min, x_max), None, (z_min, z_max))*.

AXES = {'x': [0], 'y': [1], 'z': [2], 'xy': [0, 1], 'yz': [1, 2], 'xz': [0, 2],

distance_generator(*f*, *g*)

distances(*A*, *B*, *expand=False*)

Calculate the distance matrix between two sets of coordinates, given the topology of the current space. From http://projects.scipy.org/pipermail/numpy-discussion/2007-April/027203.html

class spynnaker8.**Sphere**(*radius*)

Bases: pyNN.space.Shape

Represents a spherical volume within which neurons may be distributed.

sample(*n*, *rng*)

Return *n* points distributed randomly with uniform density within the sphere.

class spynnaker8.**AllToAllConnector**(*allow_self_connections=True*, *safe=True*, *verbose=None*, *callbacks=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.all_to_all_connector.AllToAllConnector, pyNN.connectors.AllToAllConnector

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** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **verbose** –
- **callbacks** –

class spynnaker8.**ArrayConnector**(*array*, *safe=True*, *callback=None*, *verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.array_connector.ArrayConnector

Create an array connector.

Parameters **array** (*integer*) – an array of integers

class spynnaker8.**CSAConnector**(*cset*, *safe=True*, *callback=None*, *verbose=False*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.csa_connector.CSAConnector

Create an CSA (Connection Set Algebra, Djurfeldt 2012) connector.

Parameters **cset** (*string*) – a connection set description

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.distance_dependent_probability_connector.DistanceDependentProbabilityConnector, pyNN.connectors.DistanceDependentProbabilityConnector

Make connections using a distribution which varies with distance.

Parameters

- **d_expression** (*string*) – 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.
- **n_connections** (*int*) – The number of efferent synaptic connections per neuron.
- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.

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.fixed_number_post_connector.FixedNumberPostConnector, pyNN.connectors.FixedNumberPostConnector

PyNN connector that puts a fixed number of connections on each of the post 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*) – 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** – random number generator
- **callback** – list of callbacks to run

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.fixed_number_pre_connector.FixedNumberPreConnector, pyNN.connectors.FixedNumberPreConnector

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 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** –
- **callback** –

class spynnaker8.**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, pyNN.connectors.FixedProbabilityConnector

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

Parameters

- **p_connect** (*float*) – a number between zero and one. Each potential connection is created with this probability.
- **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.
- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **space** – a Space object, needed if you wish to specify distance-dependent weights or delays - not implemented
- **verbose** –
- **rng** –
- **callback** –

p_connect

class spynnaker8.**FromFileConnector**(*file*, *distributed=False*, *safe=True*, *callback=None*, *verbose=False*)

Bases: *spynnaker8.models.connectors.from_list_connector.FromListConnector*, pyNN.connectors.FromFileConnector

get_reader(*file*)

Get a file reader object using the PyNN methods.

Returns A pynn StandardTextFile or similar

class spynnaker8.**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.

Parameters

- **conn_list** – 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).
- **column_names** – the names of the parameters p1, p2, etc. If not provided, it is assumed the parameters are weight, delay (for backwards compatibility).
- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** – if given, a callable that display a progress bar on the terminal.

class spynnaker8.**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.

Parameters

- **index_expression** (*str*) – a function of the indices of the populations An expression
- **allow_self_connections** (*bool*) – allow a neuron to connect to itself

spynnaker8.**FixedTotalNumberConnector**

alias of *spynnaker8.models.connectors.multapse_connector.MultapseConnector*

class spynnaker8.**OneToOneConnector**(*safe=True*, *callback=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.one_to_one_connector.OneToOneConnector, pyNN.connectors.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.

Parameters

- **safe** – if True, check that weights and delays have valid values. If False, this check is skipped.
- **callback** – a function that will be called with the fractional progress of the connection routine. An example would be progress_bar.set_level.

class spynnaker8.**SmallWorldConnector**(*degree*, *rewiring*, *allow_self_connections=True*, *safe=True*, *verbose=False*, *n_connections=None*)

Bases: spynnaker.pyNN.models.neural_projections.connectors.small_world_connector.SmallWorldConnector

class spynnaker8.**KernelConnector**(*shape_pre*, *shape_post*, *shape_kernel*, *weight_kernel=None*, *delay_kernel=None*, *shape_common=None*, *pre_sample_steps=None*, *pre_start_coords=None*, *post_sample_steps=None*, *post_start_coords=None*, *safe=True*, *space=None*, *verbose=False*)

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.

TODO: should these include allow_self_connections and with_replacement?

Parameters

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

spynnaker8.**StaticSynapse**
alias of *spynnaker8.models.synapse_dynamics.synapse_dynamics_static.SynapseDynamicsStatic*

spynnaker8.**STDPMechanism**
alias of *spynnaker8.models.synapse_dynamics.synapse_dynamics_stdp.SynapseDynamicsSTDP*

spynnaker8.**AdditiveWeightDependence**
alias of *spynnaker8.models.synapse_dynamics.weight_dependence.weight_dependence_additive.WeightDependenceAdditive*

spynnaker8.**MultiplicativeWeightDependence**
alias of *spynnaker8.models.synapse_dynamics.weight_dependence.weight_dependence_multiplicative.WeightDependenceMultiplicative*

spynnaker8.**SpikePairRule**
alias of *spynnaker8.models.synapse_dynamics.timing_dependence.timing_dependence_spike_pair.TimingDependenceSpikePair*

spynnaker8.**StructuralMechanismStatic**
alias of *spynnaker8.models.synapse_dynamics.synapse_dynamics_structural_static.SynapseDynamicsStructuralStatic*

spynnaker8.**StructuralMechanismSTDP**
alias of *spynnaker8.models.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

vertex_executable_suffix

The suffix to be appended to the vertex executable for this rule

write_parameters(*spec*)

Write the parameters of the rule to the spec

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

vertex_executable_suffix

The suffix to be appended to the vertex executable for this rule

write_parameters(*spec*)

Write the parameters of the rule to the spec

class spynnaker8.**DistanceDependentFormation**(*grid=array([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** – (x, y) dimensions of the grid of distance
- **p_form_forward** – The peak probability of formation on feed-forward connections
- **sigma_form_forward** – The spread of probability with distance of formation on feed-forward connections
- **p_form_lateral** – The peak probability of formation on lateral connections
- **sigma_form_lateral** – 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** (*np.ndarray of ints*) – first point in space
- **x1** (*np.ndarray of ints*) – second point in space
- **grid** (*np.ndarray of ints*) – shape of grid
- **metric** (*str*) – distance metric, i.e. euclidian or manhattan

Returns the distance

Return type float

generate_distance_probability_array(*probability*, *sigma*)
Generate the exponentially decaying probability LUTs.

Parameters

- **probability** (*float*) – peak probability
- **sigma** (*float*) – spread

Returns distance-dependent probabilities

Return type numpy.ndarray(float)

get_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

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters(*spec*)
Write the parameters of the rule to the spec

class spynnaker8.**RandomByWeightElimination**(*threshold*, *prob_elim_depressed=0.0245*, *prob_elim_potentiatiated=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** – 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** – The probability of elimination if the weight has been depressed (ignored on static weight connections)
- **prob_elim_potentiatiated** – 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

vertex_executable_suffix
The suffix to be appended to the vertex executable for this rule

write_parameters(*spec*, *weight_scale*)
Write the parameters of the rule to the spec

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.**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*)
Create a vertex for a population of the model

Parameters

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

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {}

class 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*)
Create a vertex for a population of the model

Parameters

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

Returns An application vertex for the population

Return type pacman.model.graphs.application.ApplicationVertex

default_population_parameters = {'max_rate': None, 'seed': None}

classmethod get_max_atoms_per_core()
Get the maximum number of atoms per core for this model

Return type int

classmethod set_model_max_atoms_per_core(*n_atoms=500*)
Set the maximum number of atoms per core for this model

Parameters **n_atoms** (*int or None*) – The new maximum, or None for the largest possible

class spynnaker8.**Assembly**(**populations*, ***kwargs*)
Bases: pyNN.common.populations.Assembly

Create an Assembly of Populations and/or PopulationViews.

class spynnaker8.**Population**(*size*, *cellclass*, *cellparams=None*, *structure=None*, *initial_values=None*, *label=None*, *constraints=None*, *additional_parameters=None*)
Bases: spynnaker.pyNN.models.pynn_population_common.PyNNPopulationCommon, *spynnaker8.models.recorder.Recorder*, *spynnaker8.models.populations.population_base.PopulationBase*

PyNN 0.8/0.9 population object

all()
Iterator over cell IDs on all MPI nodes.

all_cells
An array containing the cell IDs of all neurons in the Population (all MPI nodes).

annotations
The annotations given by the end user

can_record(*variable*)
Determine whether *variable* can be recorded from this population.

celltype
Implements the PyNN expected celltype property

Returns The celltype this property has been set to

static create(*cellclass*, *cellparams=None*, *n=1*)
Pass through method to the constructor defined by PyNN. Create n cells all of the same type. Returns a Population object.

Parameters

- **cellclass** – see Population.__init__
- **cellparams** – see Population.__init__
- **n** – see Population.__init__(size...)

Returns A New 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.

find_units(*variable*)
Get the units of a variable

Parameters **variable** – The name of the variable

Returns The units of the variable

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*) – 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*) – annotations to put on the neo block

Return type neo.Block

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*) – 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 get_data
- **clear** (*bool*) – Whether recorded data will be deleted.
- **annotations** (*dict*) – annotations to put on the neo block

Return type neo.Block

get_initial_value(*variable*, *selector=None*)
See AbstractPopulationInitializable.get_initial_value

get_initial_values(*selector=None*)
See AbstractPopulationInitializable.get_initial_values

get_spike_counts(*gather=True*)
Return the number of spikes for each neuron.

initial_values

initialize(***kwargs*)

position_generator
NO PyNN description of this method.

positions
Return the position array for structured populations.

record(*variables*, *to_file=None*, *sampling_interval=None*, *indexes=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** (*a Neo IO instance*) – 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.
- **indexes** – The indexes of neurons to record from. This is none Standard PyNN and equivelent to creating a view with these indexes and asking the View to record.

sample(*n*, *rng=None*)
Randomly sample *n* cells from the Population, and return a PopulationView object.

set(***parameters*)
Set one or more parameters for every cell in the population.

param can be a dict, in which case value should not be supplied, or a string giving the parameter name, in which case value is the parameter value. value can be a numeric value, or list of such (e.g. for setting spike times):

```
p.set("tau_m", 20.0).
p.set({'tau_m':20, 'v_rest':-65})
```

Parameters

- **parameter** (*str or dict*) – the parameter to set
- **value** – the value of the parameter to set.

set_initial_value(*variable*, *value*, *selector=None*)
See AbstractPopulationInitializable.set_initial_value

spinnaker_get_data(*variable*)
Public accessor for getting data as a numpy array, instead of the neo based object

Parameters **variable** – either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.

Returns numpy array of the data

tset(***kwargs*)
Deprecated. 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 instance 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** – pointless on sPyNNaker
- **clear** – clears the storage data if set to true after reading it back
- **annotations** – annotations to put on the neo block

class spynnaker8.**PopulationView**(*parent*, *selector*, *label=None*)
Bases: *spynnaker8.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 **selector** – 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.

all()
Iterator over cell IDs (on all MPI nodes).

all_cells
An array containing the cell IDs of all neurons in the Population (all MPI nodes).

can_record(*variable*)
Determine whether variable can be recorded from this population.

celltype
The type of neurons making up the Population.

conductance_based
Indicates whether the post-synaptic response is modelled as a change in conductance or a change in current.

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.

find_units(*variable*)

Warning: NO PyNN description of this method.

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).

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** – Either a single variable name or a list of variable names. Variables must have been previously recorded, otherwise an Exception will be raised.
- **gather** – 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** – If True, recorded data will be deleted from the Population. :param annotations: annotations to put on the neo block

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 tbe Populations uses PyNN 7 version of the get_spikes method which dooes not support indexes.

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.

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

index_in_grandparent(*indices*)

Given an array of indices, return the indices in the parent population at the root of the tree.

initial_values

A dict containing the initial values of the state variables.

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, 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)
```

label

A label for the Population.

mask
The selector mask that was used to create this view.

parent
A reference to the parent Population (that this is a view of).

record(*variables*, *to_file=None*, *sampling_interval=None*)
Record the specified variable or variables for all cells in the Population or view.

Parameters

- **varables** – 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** – If specified, should be a Neo IO instance and write_data() will be automatically called when end() is called.
- **sampling_interval** – 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, and return a PopulationView object.

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.

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** – a Neo IO instance
- **variables** – either a single variable name or a list of variable names. These must have been previously recorded, otherwise an Exception will be raised.
- **gather** – 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.
- **clear** – If this is True, recorded data will be deleted from the Population.
- **annotations** – 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.

spynnaker8.**SpiNNakerProjection**
alias of *spynnaker8.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

Return type None

spynnaker8.**setup**(*timestep=0.1*, *min_delay='auto'*, *max_delay='auto'*, *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** – the time step of the simulations
- **min_delay** – the min delay of the simulation
- **max_delay** – the max delay of the simulation
- **graph_label** – the label for the graph
- **database_socket_addresses** – the sockets used by external devices for the database notification protocol
- **extra_algorithm_xml_paths** – list of paths to where other XML are located
- **extra_mapping_inputs** – other inputs used by the mapping process
- **extra_mapping_algorithms** – other algorithms to be used by the mapping process
- **extra_pre_run_algorithms** – extra algorithms to use before a run
- **extra_post_run_algorithms** – extra algorithms to use after a run
- **extra_load_algorithms** – extra algorithms to use within the loading phase
- **time_scale_factor** – 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** – 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 stuff

Returns rank thing

: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** – time to run for (in milliseconds)

- **callbacks** – callbacks to run

Returns the actual simulation time that the simulation stopped at

spynnaker8.**run_until**(*tstop*)
Run until a (simulation) time period has completed.

Parameters **tstop** – the time to stop at (in milliseconds)

Returns the actual simulation time that the simulation stopped at

spynnaker8.**run_for**(*simtime*, *callbacks=None*)
The run() function advances the simulation for a given number of milliseconds, e.g.:

Parameters

- **simtime** – time to run for (in milliseconds)
- **callbacks** – callbacks to run

Returns the actual simulation time that the simulation stopped at

spynnaker8.**num_processes**()
The number of MPI processes.

Note: Always 1 on SpiNNaker, which doesn't use MPI.

Returns the number of MPI processes

spynnaker8.**rank**()
The MPI rank of the current node.

Note: Always 0 on SpiNNaker, whcih doesn't use MPI.

Returns MPI rank

spynnaker8.**reset**(*annotations=None*)
Resets the simulation to t = 0

Parameters **annotations** – 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** – neuron type
- **max_permitted** – the number to set to

Return type None

spynnaker8.**get_projections_data**(*projection_data*)

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** – the source pop
- **postsynaptic_population** – the dest pop
- **connector** – the connector type
- **synapse_type** – the synapse type
- **source** – the source
- **receptor_type** – the recpetor type
- **space** – the space object
- **label** – the label

Returns a projection object for SpiNNaker

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** – population class
- **cellparams** – population params.
- **n** – n neurons

Return type None

spynnaker8.**connect**(*pre*, *post*, *weight=0.0*, *delay=None*, *receptor_type=None*, *p=1*, *rng=None*)
Builds a projection

Parameters

- **pre** – source pop
- **post** – destination pop
- **weight** – weight of the connections
- **delay** – the delay of the connections
- **receptor_type** – excitatory / inhibitatory
- **p** – probability
- **rng** – random number generator

Return type None

spynnaker8.**get_time_step**()
The integration time step

Returns get the time step of the simulation (in ms)

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

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

spynnaker8.**initialize**(*cells*, ***initial_values*)
Sets cells to be initialised to the given values

Parameters

- **cells** – the cells to change params on
- **initial_values** – the params and there values to change

Return type None

spynnaker8.**list_standard_models**()
Return a list of all the StandardCellType classes available for this simulator.

spynnaker8.**name**()
Returns the name of the simulator

:rtype:None

spynnaker8.**num_processes**()
The number of MPI processes.

Note: Always 1 on SpiNNaker, which doesn't use MPI.

Returns the number of MPI processes

spynnaker8.**record**(*variables*, *source*, *filename*, *sampling_interval=None*, *annotations=None*)
Sets variables to be recorded.

Parameters

- **variables** – 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** – where to record from
- **filename** – file name to write data to
- **sampling_interval** – how often to sample the recording, not ignored so far
- **annotations** – the annotations to data writers

Returns neo object

spynnaker8.**record_v**(*source*, *filename*)
Deprecated method for getting voltage. This is not documented in the public facing API.

Parameters

- **source** – the population / view / assembly to record
- **filename** – 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

Parameters

- **source** – the population / view / assembly to record
- **filename** – the neo file to write to

Return type None

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

A

B

C

D

E

F

G

H

I

K

L

M

Q

R

S

U

V

W