GeNN

Sep 05, 2019

Contents

GeNN is a software package to enable neuronal network simulations on NVIDIA GPUs by code generation. *Models* are defined in a simple C-style API and the code for running them on either GPU or CPU hardware is generated by GeNN. GeNN can also be used through external interfaces. Currently there are interfaces for *SpineML and SpineCreator* and for Brian via *Brian2GeNN*.

GeNN is currently developed and maintained by

Dr James Knight (contact James)

James Turner (contact James)

Prof. Thomas Nowotny (contact Thomas)

Project homepage is http://genn-team.github.io/genn/.

The development of GeNN is partially supported by the EPSRC (grant numbers EP/P006094/1 - Brains on Board and EP/J019690/1 - Green Brain Project).

This documentation is under construction. If you cannot find what you are looking for, please contact the project developers.

Next

CHAPTER 1

Best practices guide

GeNN generates code according to the network model defined by the user, and allows users to include the generated code in their programs as they want. Here we provide a guideline to setup GeNN and use generated functions. We recommend users to also have a look at the *Examples*, and to follow the tutorials *Tutorial 1* and *Tutorial 2*.

1.1 Creating and simulating a network model

The user is first expected to create an object of class *ModelSpec* by creating the function modelDefinition() which includes calls to following methods:

- *ModelSpec::setDT()*;
- *ModelSpec::setName()*;

Then add neuron populations by:

- *ModelSpec::addNeuronPopulation()*;

for each neuron population. Add synapse populations by:

- *ModelSpec::addSynapsePopulation()*;

for each synapse population.

Other optional functions are explained in *ModelSpec* class reference. At the end the function should look like this:

```
void modelDefinition(ModelSpec &model)
  model.setDT(0.5);
  model.setName("YourModelName");
  model.addNeuronPopulation(...);
  ...
  model.addSynapsePopulation(...);
  ...
```

`modelSpec.h` should be included in the file where this function is defined.

This function will be called by generateALL.cc to create corresponding CPU and GPU simulation codes under the <YourModelName>_CODE directory.

These functions can then be used in a .cc file which runs the simulation. This file should include <YourModelName>_CODE/definitions.h. Generated code differ from one model to the other, but core functions are the same and they should be called in correct order. First, the following variables should be defined and initialized:

- *ModelSpec* model // initialized by calling modelDefinition(model)
- Array containing current input (if any)

Any variables marked as unintialised using the `uninitialisedVar()` function or sparse connectivity not initialised using a snippet must be initialised by the user between calls to `initialize()` and `initializeSparse()`. Core functions generated by GeNN to be included in the user code include:

- `allocateMem()`
- `initialize()`
- `initializeSparse()`
- `push<neuron or synapse name>StateToDevice()`
- `pull<neuron or synapse name>StateFromDevice()`
- `push<neuron name>SpikesToDevice()`
- `pull<neuron name>SpikesFromDevice()`
- `push<neuron name>SpikesEventsToDevice()`
- `pull<neuron name>SpikesEventsFromDevice()`
- `push<neuron name>SpikeTimesToDevice()`
- `pull<neuron name>SpikeTimesFromDevice()`
- `push<neuron name>CurrentSpikesToDevice()`
- `pull<neuron name>CurrentSpikesFromDevice()`
- `push<neuron name>CurrentSpikesEventsToDevice()`
- `pull<neuron name>CurrentSpikesEventsFromDevice()`
- `pull<synapse name>ConnectivityFromDevice()`
- `push<synapse name>ConnectivityToDevice()`
- `pull<var name><neuron or synapse name>FromDevice()`
- `push<var name><neuron or synapse name>ToDevice()`
- `copyStateToDevice()`
- `copyStateFromDevice()`
- `copyCurrentSpikesFromDevice()`
- `copyCurrentSpikesEventsFromDevice()`
- `stepTime()`
- `freeMem()`

You can use the `push<neuron or synapse name>StateToDevice()` to copy from the host to the GPU. At the end of your simulation, if you want to access the variables you need to copy them back from the device using the `pull<neuron or synapse name>StateFromDevice()` function or one of the more fine-grained functions listed above. **Copying elements between the GPU and the host memory is very costly in terms of performance and should only be done when needed and the amount of data being copied should be minimized.**

1.1.1 Extra Global Parameters

If extra global parameters have a "scalar" type such as `float` they can be set directly from simulation code. For example the extra global parameter "reward" of population "Pop" could be set with:

```
rewardPop = 5.0f;
```

However, if extra global parameters have a pointer type such as `float*`, GeNN generates additional functions to allocate, free and copy these variables between host and device:

- `allocate<var name><neuron or synapse name>`
- `free<var name><neuron or synapse name>`
- `push<var name><neuron or synapse name>ToDevice`
- `pull<var name><neuron or synapse name>FromDevice` These operate in much the same manner as the functions for interacting with standard variables described above but the allocate, push and pull functions all take a "count" parameter specifying how many entries the extra global parameter array should be.

1.2 Floating point precision

Double precision floating point numbers are supported by devices with compute capability 1.3 or higher. If you have an older GPU, you need to use single precision floating point in your models and simulation.

GPUs are designed to work better with single precision while double precision is the standard for CPUs. This difference should be kept in mind while comparing performance.

While setting up the network for GeNN, double precision floating point numbers are used as this part is done on the CPU. For the simulation, GeNN lets users choose between single or double precision. Overall, new variables in the generated code are defined with the precision specified by ModelSpec::setPrecision(unsigned int), providing GENN_FLOAT or GENN_DOUBLE as argument. GENN_FLOAT is the default value. The keyword `scalar` can be used in the user-defined model codes for a variable that could either be single or double precision. This keyword is detected at code generation and substituted with "float" or "double" according to the precision set by ModelSpec::setPrecision(unsigned int).

There may be ambiguities in arithmetic operations using explicit numbers. Standard C compilers presume that any number defined as "X" is an integer and any number defined as "X.Y" is a double. Make sure to use the same precision in your operations in order to avoid performance loss.

1.3 Working with variables in GeNN

1.3.1 Model variables

User-defined model variables originate from classes derived off the *NeuronModels::Base*, *WeightUpdateModels::Base* or *PostsynapticModels::Base* classes. The name of model variable is defined in the model type, i.e. with a statement such as

```
SET_VARS("V", "scalar");
```

When a neuron or synapse population using this model is added to the model, the full GeNN name of the variable will be obtained by concatenating the variable name with the name of the population. For example if we a add a population called `Pop` using a model which contains our `V` variable, a variable `VPop` of type `scalar*` will be available in the global namespace of the simulation program. GeNN will pre-allocate this C array to the correct size of elements corresponding to the size of the neuron population. GeNN will also free these variables when the provided function `freeMem()` is called. Users can otherwise manipulate these variable arrays as they wish. For convenience, GeNN provides functions to copy each state variable from the device into host memory and vice versa e.g. `pullVPopFromDevice()` and `pushVPoptoDevice()`. Alternatively, all state variables associated with a population can be copied using a single call E.g.

```
pullPopStateFromDevice();
```

These conventions also apply to the the variables of postsynaptic and weight update models. Be aware that the above naming conventions do assume that variables from the weightupdate models and the postSynModels that are used together in a synapse population are unique. If both the weightupdate model and the postSynModel have a variable of the same name, the behaviour is undefined.

1.3.2 Built-in Variables in GeNN

GeNN has no explicitly hard-coded synapse and neuron variables. Users are free to name the variable of their models as they want. However, there are some reserved variables that are used for intermediary calculations and communication between different parts of the generated code. They can be used in the user defined code but no other variables should be defined with these names.

- `DT` : Time step (typically in ms) for simulation; Neuron integration can be done in multiple sub-steps inside the neuron model for numerical stability (see Traub-Miles and Izhikevich neuron model variations in *Neuron models*).
- `inSyn`: This is an intermediary synapse variable which contains the summed input into a postsynaptic neuron (originating from the `addtoinSyn` variables of the incoming synapses) .
- `Isyn` : This is a local variable which contains the (summed) input current to a neuron. It is typically the sum of any explicit current input and all synaptic inputs. The way its value is calculated during the update of the postsynaptic neuron is defined by the code provided in the postsynaptic model. For example, the standard *PostsynapticModels::ExpCond* postsynaptic model defines

  ```
  SET_APPLY_INPUT_CODE("$(Isyn) += $(inSyn)*($(E)-$(V))");
  ```

  which implements a conductance based synapse in which the postsynaptic current is given by $I_{\text{syn}} = g * s * (V_{\text{rev}} - V_{\text{post}})$.

The value resulting from the current converter code is assigned to `Isyn` and can then be used in neuron sim code like so:

```
$(V)+= (-$(V)+$(Isyn))*DT
```

- `sT` : This is a neuron variable containing the last spike time of each neuron and is automatically generated for pre and postsynaptic neuron groups if they are connected using a synapse population with a weight update model that has SET_NEEDS_PRE_SPIKE_TIME(true) or SET_NEEDS_POST_SPIKE_TIME(true) set.

In addition to these variables, neuron variables can be referred to in the synapse models by calling $(<neuronVarName>_pre) for the presynaptic neuron population, and $(<neuronVarName>_post) for the postsynaptic population. For example, $(sT_pre), $(sT_post), $(V_pre), etc.

1.4 Debugging suggestions

In Linux, users can call `cuda-gdb` to debug on the GPU. Example projects in the `userproject` directory come with a flag to enable debugging (debug). genn-buildmodel.sh has a debug flag (-d) to generate debugging data. If you are executing a project with debugging on, the code will be compiled with -g -G flags. In CPU mode the executable will be run in gdb, and in GPU mode it will be run in cuda-gdb in tui mode.

Do not forget to switch debugging flags -g and -G off after debugging is complete as they may negatively affect performance.

On Mac, some versions of `clang` aren't supported by the CUDA toolkit. This is a recurring problem on Fedora as well, where CUDA doesn't keep up with GCC releases. You can either hack the CUDA header which checks compiler versions - `cuda/include/host_config.h` - or just use an older XCode version (6.4 works fine).

On Windows models can also be debugged and developed by opening the sln file used to build the model in Visual Studio. From here files can be added to the project, build settings can be adjusted and the full suite of Visual Studio debugging and profiling tools can be used. When opening the models in the `userproject` directory in Visual Studio, right-click on the project in the solution explorer, select 'Properties'. Then, making sure the desired configuration is selected, navigate to 'Debugging' under 'Configuration Properties', set the 'Working Directory' to '..' and the 'Command Arguments' to match those passed to genn-buildmodel e.g. 'outdir' to use an output directory called outdir.

Previous | *Top* | *Next*

CHAPTER 2

Bibliographic References

CHAPTER 3

Brian interface (Brian2GeNN)

GeNN can simulate models written for the Brian simulator via the Brian2GeNN interface brian2genn2018. The easiest way to install everything needed is to install the Anaconda or Miniconda Python distribution and then follow the instructions to install Brian2GeNN with the conda package manager. When Brian2GeNN is installed in this way, it comes with a bundled version of GeNN and no further configuration is required. In all other cases (e.g. an installation from source), the path to GeNN and the CUDA libraries has to be configured via the `GENN_PATH` and `CUDA_PATH` environment variables as described in *Installation* or via the `devices.genn.path` and `devices.genn.cuda_path` Brian preferences.

To use GeNN to simulate a Brian script, import the `brian2genn` package and switch Brian to the `genn` device. As an example, the following Python script will simulate Leaky-integrate-and-fire neurons with varying input currents to construct an f/I curve:

```
from brian2 import *
import brian2genn
set_device('genn')

n = 1000
duration = 1*second
tau = 10*ms
eqs = '''
dv/dt = (v0 - v) / tau : volt (unless refractory)
v0 : volt
'''
group = NeuronGroup(n, eqs, threshold='v > 10*mV', reset='v = 0*mV',
                    refractory=5*ms, method='exact')
group.v = 0*mV
group.v0 = '20*mV * i / (n-1)'
monitor = SpikeMonitor(group)

run(duration)
```

Of course, your simulation should be more complex than the example above to actually benefit from the performance gains of using a GPU via GeNN. *Previous* | *Top* | *Next*

CHAPTER 4

Credits

GeNN was created by Thomas Nowotny.

GeNN is currently maintained and developed by James Knight.

Current sources and PyGeNN were first implemented by Anton Komissarov.

Izhikevich model and sparse connectivity by Esin Yavuz.

Block size optimisations, delayed synapses and page-locked memory by James Turner.

Automatic brackets and dense-to-sparse network conversion helper tools by Alan Diamond.

User-defined synaptic and postsynaptic methods by Alex Cope and Esin Yavuz.

Example projects were provided by Alan Diamond, James Turner, Esin Yavuz and Thomas Nowotny.

MPI support was largely developed by Mengchi Zhang.

Previous

CHAPTER 5

Examples

GeNN comes with a number of complete examples. At the moment, there are seven such example projects provided with GeNN.

5.1 Single compartment Izhikevich neuron(s)

```
Izhikevich neuron(s) without any connections
====================================

This is a minimal example, with only one neuron population (with more or less
neurons depending on the command line, but without any synapses). The neurons
are Izhikevich neurons with homogeneous parameters across the neuron population.
This example project contains a helper executable called "generate_run",
which compiles and executes the model.

To compile it, navigate to genn/userproject/OneComp_project and type:

msbuild ..\userprojects.sln /t:generate_one_comp_runner /p:Configuration=Release

for Windows users, or:

make

for Linux, Mac and other UNIX users.


USAGE
-----

generate_run [OPTIONS] <outname>

Mandatory arguments:
outname: The base name of the output location and output files
```

(continues on next page)

(continued from previous page)

```
Optional arguments:
--debug: Builds a debug version of the simulation and attaches the debugger
--cpu-only: Uses CPU rather than CUDA backend for GeNN
--timing: Uses GeNN's timing mechanism to measure performance and displays it at the␣
↪end of the simulation
--ftype: Sets the floating point precision of the model to either float or double␣
↪(defaults to float)
--gpu-device: Sets which GPU device to use for the simulation (defaults to -1 which␣
↪picks automatically)
--num-neurons: Number of neurons to simulate (defaults to 1)

For a first minimal test, using these defaults and recording results with a base name␣
↪of `test',the system may be used with:

generate_run.exe test

for Windows users, or:

./generate_run test

for Linux, Mac and other UNIX users.

This would create a set of tonic spiking Izhikevich neurons with no connectivity,
receiving a constant identical 4 nA input.

Another example of an invocation that runs the simulation using the CPU rather than␣
↪GPU,
records timing information and 4 neurons would be:

generate_run.exe --cpu-only --timing --num_neurons=4 test

for Windows users, or:

./generate_run --cpu-only --timing --num_neurons=4 test
```

Izhikevich neuron model: izhikevich2003simple

5.2 Izhikevich neurons driven by Poisson input spike trains:

```
Izhikevich network receiving Poisson input spike trains
=======================================================

In this example project there is again a pool of non-connected Izhikevich model␣
↪neurons
that are connected to a pool of Poisson input neurons with a fixed probability.
This example project contains a helper executable called "generate_run", which␣
↪compiles and
executes the model.

To compile it, navigate to genn/userproject/PoissonIzh_project and type:

msbuild ..\userprojects.sln /t:generate_poisson_izh_runner /p:Configuration=Release
```

(continues on next page)

(continued from previous page)

```
for Windows users, or:

make

for Linux, Mac and other UNIX users.


USAGE
-----

generate_run [OPTIONS] <outname>

Mandatory arguments:
outname: The base name of the output location and output files

Optional arguments:
--debug: Builds a debug version of the simulation and attaches the debugger
--cpu-only: Uses CPU rather than CUDA backend for GeNN
--timing: Uses GeNN's timing mechanism to measure performance and displays it at the␣
↪end of the simulation
--ftype: Sets the floating point precision of the model to either float or double␣
↪(defaults to float)
--gpu-device: Sets which GPU device to use for the simulation (defaults to -1 which␣
↪picks automatically)
--num-poisson:  Number of Poisson sources to simulate (defaults to 100)
--num-izh: Number of Izhikievich neurons to simulate (defaults to 10)
--pconn: Probability of connection between each pair of poisson sources and neurons␣
↪(defaults to 0.5)
--gscale: Scaling of synaptic conductances (defaults to 2)
--sparse: Use sparse rather than dense data structure to represent connectivity

An example invocation of generate_run using these defaults and recording results with␣
↪a base name of `test':

generate_run.exe test

for Windows users, or:

./generate_run test

for Linux, Mac and other UNIX users.

This will generate a network of 100 Poisson neurons with 20 Hz firing rate
connected to 10 Izhikevich neurons with a 0.5 probability.
The same network with sparse connectivity can be used by adding the --sparse flag to␣
↪the command line.

Another example of an invocation that runs the simulation using the CPU rather than␣
↪GPU,
records timing information and uses sparse connectivity would be:

generate_run.exe --cpu-only --timing --sparse test

for Windows users, or:

./generate_run --cpu-only --timing --sparse test
```

(continues on next page)

(continued from previous page)

Izhikevich neuron model: izhikevich2003simple

5.3 Pulse-coupled Izhikevich network

```
Pulse-coupled Izhikevich network
================================

This example model is inspired by simple thalamo-cortical network of Izhikevich
with an excitatory and an inhibitory population of spiking neurons that are
randomly connected. It creates a pulse-coupled network with 80% excitatory 20%
inhibitory connections, each connecting to a fixed number of neurons with sparse␣
↪connectivity.

To compile it, navigate to genn/userproject/Izh_sparse_project and type:

msbuild ..\userprojects.sln /t:generate_izh_sparse_runner /p:Configuration=Release

for Windows users, or:

make

for Linux, Mac and other UNIX users.


USAGE
-----

generate_run [OPTIONS] <outname>

Mandatory arguments:
outname: The base name of the output location and output files

Optional arguments:
--debug: Builds a debug version of the simulation and attaches the debugger
--cpu-only: Uses CPU rather than CUDA backend for GeNN
--timing: Uses GeNN's timing mechanism to measure performance and displays it at the␣
↪end of the simulation
--ftype: Sets the floating point precision of the model to either float or double␣
↪(defaults to float)
--gpu-device: Sets which GPU device to use for the simulation (defaults to -1 which␣
↪picks automatically)
--num-neurons: Number of neurons (defaults to 10000)
--num-connections: Number of connections per neuron (defaults to 1000)
--gscale: General scaling of synaptic conductances (defaults to 1.0)

An example invocation of generate_run using these defaults and recording results with␣
↪a base name of `test' would be:

generate_run.exe test

for Windows users, or:
```

(continues on next page)

(continued from previous page)

```
./generate_run test

for Linux, Mac and other UNIX users.

This would create a pulse coupled network of 8000 excitatory 2000 inhibitory
Izhikevich neurons, each making 1000 connections with other neurons, generating
a mixed alpha and gamma regime. For larger input factor, there is more
input current and more irregular activity, for smaller factors less
and less and more sparse activity. The synapses are of a simple pulse-coupling
type. The results of the simulation are saved in the directory `outdir_output`.

Another example of an invocation that runs the simulation using the CPU rather than␣
↪GPU,
records timing information and doubles the number of neurons would be:

generate_run.exe --cpu-only --timing --num_neurons=20000 test

for Windows users, or:

./generate_run --cpu-only --timing --num_neurons=20000 test
```

Izhikevich neuron model: izhikevich2003simple

5.4 Izhikevich network with delayed synapses

```
Izhikevich network with delayed synapses
========================================

This example project demonstrates the synaptic delay feature of GeNN. It creates
a network of three Izhikevich neuron groups, connected all-to-all with fast, medium
and slow synapse groups. Neurons in the output group only spike if they are
simultaneously innervated by the input neurons, via slow synapses, and the
interneurons, via faster synapses.


COMPILE (WINDOWS)
-----------------

To run this example project, first build the model into CUDA code by typing:

genn-buildmodel.bat SynDelay.cc

then compile the project by typing:

msbuild SynDelay.sln /t:SynDelay /p:Configuration=Release


COMPILE (MAC AND LINUX)
-----------------------

To run this example project, first build the model into CUDA code by typing:

genn-buildmodel.sh SynDelay.cc
```

(continues on next page)

(continued from previous page)

```
then compile the project by typing:

make


USAGE
-----
```

Izhikevich neuron model: izhikevich2003simple

5.5 Insect olfaction model

```
Locust olfactory system (Nowotny et al. 2005)
=============================================

This project implements the insect olfaction model by Nowotny et
al. that demonstrates self-organized clustering of odours in a
simulation of the insect antennal lobe and mushroom body. As provided
the model works with conductance based Hodgkin-Huxley neurons and
several different synapse types, conductance based (but pulse-coupled)
excitatory synapses, graded inhibitory synapses and synapses with a
simplified STDP rule. This example project contains a helper executable called
↪"generate_run", which
prepares input pattern data, before compiling and
executing the model.

To compile it, navigate to genn/userproject/MBody1_project and type:

msbuild ..\userprojects.sln /t:generate_mbody1_runner /p:Configuration=Release

for Windows users, or:

make

for Linux, Mac and other UNIX users.


USAGE
-----

generate_run [OPTIONS] <outname>

Mandatory arguments:
outname: The base name of the output location and output files

Optional arguments:
--debug: Builds a debug version of the simulation and attaches the debugger
--cpu-only: Uses CPU rather than CUDA backend for GeNN
--timing: Uses GeNN's timing mechanism to measure performance and displays it at the␣
↪end of the simulation
--ftype: Sets the floating point precision of the model to either float or double␣
↪(defaults to float)
```

(continues on next page)

(continued from previous page)

```
--gpu-device: Sets which GPU device to use for the simulation (defaults to -1 which␣
↪picks automatically)
--num-al: Number of neurons in the antennal lobe (AL), the input neurons to this␣
↪model (defaults to 100)
--num-kc: Number of Kenyon cells (KC) in the "hidden layer" (defaults to 1000)
--num-lhi: Number of lateral horn interneurons, implementing gain control (defaults␣
↪to 20)
--num-dn: Number of decision neurons (DN) in the output layer (defaults to 100)
--gscale: A general rescaling factor for synaptic strength (defaults to 0.0025)
--bitmask: Use bitmasks to represent sparse PN->KC connectivity rather than dense␣
↪connectivity
--delayed-synapses: Rather than use constant delays of DT throughough, use delays of␣
↪(5 * DT) ms on KC->DN and of (3 * DT) ms on DN->DN synapse populations

An example invocation of generate_run using these defaults and recording results with␣
↪a base name of `test' would be:

generate_run.exe test

for Windows users, or:

./generate_run test

for Linux, Mac and other UNIX users.

Such a command would generate a locust olfaction model with 100 antennal lobe neurons,
1000 mushroom body Kenyon cells, 20 lateral horn interneurons and 100 mushroom body
output neurons, and launch a simulation of it on a CUDA-enabled GPU using single
precision floating point numbers. All output files will be prefixed with "test"
and will be created under the "test" directory. The model that is run is defined
in `model/MBody1.cc`, debugging is switched off and the model would be simulated using
float (single precision floating point) variables.

In more details, what generate_run program does is:
a) use another tools to generate input patterns.

b) build the source code for the model by writing neuron numbers into
   ./model/sizes.h, and executing "genn-buildmodel.sh ./model/MBody1.cc.

c) compile the generated code by invoking "make clean && make"
   running the code, e.g. "./classol_sim r1".

Another example of an invocation that runs the simulation using the CPU rather than␣
↪GPU,
records timing information and uses bitmask connectivity would be:

generate_run.exe --cpu-only --timing --bitmask test

for Windows users, or:

./generate_run --cpu-only --timing --bitmask test

for Linux, Mac and other UNIX users.

As provided, the model outputs  `test.dn.st', `test.kc.st', `test.lhi.st' and `test.
↪pn.st' files which contain
the spiking activity observed in each population inthe simulation, There are two
```

(continues on next page)

(continued from previous page)

```
columns in this ASCII file, the first one containing the time of
a spike and the second one the ID of the neuron that spiked. Users
of matlab can use the scripts in the `matlab` directory to plot
the results of a simulation and users of python can use the plot_spikes.py script in␣
↪userproject/python.
For more about the model itself and the scientific insights gained from it see␣
↪Nowotny et al. referenced below.


MODEL INFORMATION
-----------------

For information regarding the locust olfaction model implemented in this example␣
↪project, see:

T. Nowotny, R. Huerta, H. D. I. Abarbanel, and M. I. Rabinovich Self-organization in␣
↪the
olfactory system: One shot odor recognition in insects, Biol Cyber, 93 (6): 436-446␣
↪(2005),
```

Nowotny insect olfaction model: nowotny2005self; Traub-Miles Hodgkin-Huxley neuron model: Traub1991

5.6 Voltage clamp simulation to estimate Hodgkin-Huxley parameters

```
Genetic algorithm for tracking parameters in a HH model cell
============================================================

This example simulates a population of Hodgkin-Huxley neuron models using GeNN and␣
↪evolves them with a simple
guided random search (simple GA) to mimic the dynamics of a separate Hodgkin-Huxley
neuron that is simulated on the CPU. The parameters of the CPU simulated "true cell"␣
↪are drifting
according to a user-chosen protocol: Either one of the parameters gNa, ENa, gKd, EKd,␣
↪gleak,
Eleak, Cmem are modified by a sinusoidal addition (voltage parameters) or factor␣
↪(conductance or capacitance) -
protocol 0-6. For protocol 7 all 7 parameters undergo a random walk concurrently.

To compile it, navigate to genn/userproject/HHVclampGA_project and type:

msbuild ..\userproject.sln /t:generate_hhvclamp_runner /p:Configuration=Release

for Windows users, or:s

make

for Linux, Mac and other UNIX users.


USAGE
-----

generate_run [OPTIONS] <outname>
```

(continues on next page)

(continued from previous page)

```
Mandatory arguments:
outname: The base name of the output location and output files

Optional arguments:
--debug: Builds a debug version of the simulation and attaches the debugger
--cpu-only: Uses CPU rather than CUDA backend for GeNN
--timing: Uses GeNN's timing mechanism to measure performance and displays it at the␣
↪end of the simulation
--ftype: Sets the floating point precision of the model to either float or double␣
↪(defaults to float)
--gpu-device: Sets which GPU device to use for the simulation (defaults to -1 which␣
↪picks automatically)
--protocol: Which changes to apply during the run to the parameters of the "true cell
↪" (defaults to -1 which makes no changes)
--num-pops: Number of neurons in the tracking population (defaults to 5000)
--total-time: Time in ms how long to run the simulation  (defaults to 1000 ms)

An example invocation of generate_run is:

generate_run.exe test1

for Windows users, or:

./generate_run test1

for Linux, Mac and other UNIX users.

This will simulate 5000 Hodgkin-Huxley neurons on the GPU which will, for 1000 ms, be␣
↪matched to a
Hodgkin-Huxley neuron. The output files will be written into a directory of the name␣
↪test1_output,
which will be created if it does not yet exist.

Another example of an invocation that records timing information for the the␣
↪simulation and runs it for 10000 ms would be:

generate_run.exe --timing --total-time 10000

for Windows users, or:

./generate_run --timing --total-time 10000
```

Traub-Miles Hodgkin-Huxley neuron model: Traub1991

5.7 A neuromorphic network for generic multivariate data classification

```
Author: Alan Diamond, University of Sussex, 2014

This project recreates using GeNN the spiking classifier design used in the paper

"A neuromorphic network for generic multivariate data classification"
 Authors: Michael Schmuker, Thomas Pfeil, Martin Paul Nawrota
```

(continues on next page)

(continued from previous page)

```
The classifier design is based on an abstraction of the insect olfactory system.
This example uses the IRIS stadard data set as a test for the classifier

BUILD / RUN INSTRUCTIONS

Install GeNN from the internet released build, following instruction on setting your␣
↪PATH etc

Start a terminal session

cd to this project directory (userproject/Model_Schmuker_2014_project)

To build the model using the GENN meta compiler type:

genn-buildmodel.sh Model_Schmuker_2014_classifier.cc

for Linux, Mac and other UNIX systems, or:

genn-buildmodel.bat Model_Schmuker_2014_classifier.cc

for Windows systems (add -d for a debug build).

You should only have to do this at the start, or when you change your actual network␣
↪model  (i.e. editing the file Model_Schmuker_2014_classifier.cc )

Then to compile the experiment plus the GeNN created C/CUDA code type:-

make

for Linux, Mac and other UNIX users (add DEBUG=1 if using debug mode), or:

msbuild Schmuker2014_classifier.vcxproj /p:Configuration=Release

for Windows users (change Release to Debug if using debug mode).

Once it compiles you should be able to run the classifier against the included Iris␣
↪dataset.

type

./experiment .

for Linux, Mac and other UNIX systems, or:

Schmuker2014_classifier .

for Windows systems.

This is how it works roughly.
The experiment (experiment.cu) controls the experiment at a high level. It mostly␣
↪does this by instructing the classifier (Schmuker2014_classifier.cu) which does the␣
↪grunt work.

So the experiment first tells the classifier to set up the GPU with the model and␣
↪synapse data.
```

(continues on next page)

(continued from previous page)

```
Then it chooses the training and test set data.

It runs through the training set , with plasticity ON , telling the classifier to run␣
↪with the specfied observation and collecting the classifier decision.

Then it runs through the test set with plasticity OFF  and collects the results in␣
↪various reporting files.

At the highest level it also has a loop where you can cycle through a list of␣
↪parameter values e.g. some threshold value for the classifier to use. It will then␣
↪report on the performance for each value. You should be aware that some parameter␣
↪changes won't actually affect the classifier unless you invoke a re-initialisation␣
↪of some sort. E.g. anything to do with VRs will require the input data cache to be␣
↪reset between values, anything to do with non-plastic synapse weights won't get␣
↪cleared down until you upload a changed set to the GPU etc.
```

Previous | *Top* | *Next*

CHAPTER 6

Installation

You can download GeNN either as a zip file of a stable release or a snapshot of the most recent stable version or the unstable development version using the Git version control system.

6.1 Downloading a release

Point your browser to https://github.com/genn-team/genn/releases and download a release from the list by clicking the relevant source code button. Note that GeNN is only distributed in the form of source code due to its code generation design. Binary distributions would not make sense in this framework and are not provided. After downloading continue to install GeNN as described in the *Installing GeNN* section below.

6.2 Obtaining a Git snapshot

If it is not yet installed on your system, download and install Git (http://git-scm.com/). Then clone the GeNN repository from Github

```
git clone https://github.com/genn-team/genn.git
```

The github url of GeNN in the command above can be copied from the HTTPS clone URL displayed on the GeNN Github page (https://github.com/genn-team/genn).

This will clone the entire repository, including all open branches. By default git will check out the master branch which contains the source version upon which the next release will be based. There are other branches in the repository that are used for specific development purposes and are opened and closed without warning.

As an alternative to using git you can also download the full content of GeNN sources clicking on the "Download ZIP" button on the bottom right of the GeNN Github page (https://github.com/genn-team/genn).

6.3 Installing GeNN

Installing GeNN comprises a few simple steps to create the GeNN development environment. While GeNN models are normally simulated using CUDA on NVIDIA GPUs, if you want to use GeNN on a machine without an NVIDIA GPU, you can skip steps v and vi and use GeNN in "CPU_ONLY" mode.

(i) If you have downloaded a zip file, unpack GeNN.zip in a convenient location. Otherwise enter the directory where you downloaded the Git repository.

(ii) Add GeNN's "bin" directory to your path, e.g. if you are running Linux or Mac OS X and extracted/downloaded GeNN to $HOME/GeNN, then you can add:

```
export PATH=$PATH:$HOME/GeNN/bin
```

to your login script (e.g. `.profile` or `.bashrc`. If you are using WINDOWS, the path should be a windows path as it will be interpreted by the Visual C++ compiler `cl`, and environment variables are best set using `SETX` in a Windows cmd window. To do so, open a Windows cmd window by typing `cmd` in the search field of the start menu, followed by the `enter` key. In the `cmd` window type:

```
setx PATH "C:;%PATH%"
```

where `C:\Users\me\GeNN` is the path to your GeNN directory.

(iv) Install the C++ compiler on the machine, if not already present. For Windows, download Microsoft Visual Studio Community Edition from https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx. When installing Visual Studio, one should select the 'Desktop development with C++' configuration' and the 'Windows 8.1 SDK' and 'Windows Universal CRT' individual components. Mac users should download and set up Xcode from https://developer.apple.com/xcode/index.html Linux users should install the GNU compiler collection gcc and g++ from their Linux distribution repository, or alternatively from https://gcc.gnu.org/index.html Be sure to pick CUDA and C++ compiler versions which are compatible with each other. The latest C++ compiler is not necessarily compatible with the latest CUDA toolkit.

(v) If your machine has a GPU and you haven't installed CUDA already, obtain a fresh installation of the NVIDIA CUDA toolkit from https://developer.nvidia.com/cuda-downloads Again, be sure to pick CUDA and C++ compiler versions which are compatible with each other. The latest C++ compiler is not necessarily compatible with the latest CUDA toolkit.

(vi) Set the `CUDA_PATH` variable if it is not already set by the system, by putting

```
export CUDA_PATH=/usr/local/cuda
```

in your login script (or, if CUDA is installed in a non-standard location, the appropriate path to the main CUDA directory). For most people, this will be done by the CUDA install script and the default value of /usr/local/cuda is fine. In Windows, CUDA_PATH is normally already set after installing the CUDA toolkit. If not, set this variable with:

```
setx CUDA_PATH C:tocuda
```

This normally completes the installation. Windows useres must close and reopen their command window to ensure variables set using `SETX` are initialised.

If you are using GeNN in Windows, the Visual Studio development environment must be set up within every instance of the CMD.EXE command window used. One can open an instance of CMD.EXE with the development environment already set up by navigating to Start - All Programs - Microsoft Visual Studio - Visual Studio Tools - x64 Native Tools Command Prompt. You may wish to create a shortcut for this tool on the desktop, for convenience.

Top | *Next*

CHAPTER 7

Python interface (PyGeNN)

As well as being able to build GeNN models and user code directly from C++, you can also access all GeNN features from Python. The `pygenn.genn_model.GeNNModel` class provides a thin wrapper around `ModelSpec` as well as providing support for loading and running simulations; and accessing their state. `SynapseGroup`, `NeuronGroup` and `CurrentSource` are similarly wrapped by the `pygenn.genn_groups.SynapseGroup`, `pygenn.genn_groups.NeuronGroup` and `pygenn.genn_groups.CurrentSource` classes respectively.

PyGeNN can be built from source on Windows, Mac and Linux following the instructions in the README file in the pygenn directory of the GeNN repository. However, if you have a relatively recent version of Python and CUDA, we recommend that you instead downloading a suitable 'wheel' from our releases page. These can then be installed using e.g. `pip install cuda10-pygenn-0.2-cp27-cp27mu-linux_x86_64.whl` for a Linux system with CUDA 10 and Python 2.7. On Windows we recommend using the Python 3 version of Anaconda.

The following example shows how PyGeNN can be easily interfaced with standard Python packages such as numpy and matplotlib to plot 4 different Izhikevich neuron regimes:

```
import numpy as np
import matplotlib.pyplot as plt
from pygenn.genn_model import GeNNModel

# Create a single-precision GeNN model
model = GeNNModel("float", "pygenn")

# Set simulation timestep to 0.1ms
model.dT = 0.1

# Initialise IzhikevichVariable parameters - arrays will be automatically␣
↪uploaded
izk_init = "V": -65.0,
           "U": -20.0,
           "a": [0.02,     0.1,    0.02,   0.02],
           "b": [0.2,      0.2,    0.2,    0.2],
           "c": [-65.0,    -65.0,  -50.0,  -55.0],
           "d": [8.0,      2.0,    2.0,    4.0]
```

```
# Add neuron populations and current source to model
pop = model.add_neuron_population("Neurons", 4, "IzhikevichVariable", ,␣
↪izk_init)
model.add_current_source("CurrentSource", "DC", "Neurons", "amp": 10.0, )

# Build and load model
model.build()
model.load()

# Create a numpy view to efficiently access the membrane voltage from Python
voltage_view = pop.vars["V"].view

# Simulate
v = None
while model.t < 200.0:
    model.step_time()
    model.pull_state_from_device("Neurons")
    v = np.copy(voltage_view) if v is None else np.vstack((v, voltage_view))

# Create plot
figure, axes = plt.subplots(4, sharex=True)

# Plot voltages
for i, t in enumerate(["RS", "FS", "CH", "IB"]):
    axes[i].set_title(t)
    axes[i].set_ylabel("V [mV]")
    axes[i].plot(np.arange(0.0, 200.0, 0.1), v[:,i])

axes[-1].set_xlabel("Time [ms]")

# Show plot
plt.show()
```

Previous | *Top* | *Next*

CHAPTER 8

Quickstart

GeNN is based on the idea of code generation for the involved GPU or CPU simulation code for neuronal network models but leaves a lot of freedom how to use the generated code in the final application. To facilitate the use of GeNN on the background of this philosophy, it comes with a number of complete examples containing both the model description code that is used by GeNN for code generation and the "user side code" to run the generated model and safe the results. Some of the example models such as the *Insect olfaction model* use an `generate_run` executable which automates the building and simulation of the model. Using these executables, running these complete examples should be achievable in a few minutes. The necessary steps are described below.

8.1 Running an Example Model

8.1.1 Unix

In order to build the `generate_run` executable as well as any additional tools required for the model, open a shell and navigate to the `userproject/MBody1_project` directory. Then type

```
make
```

to generate an executable that you can invoke with

```
./generate_run test1
```

or, if you don't have an NVIDIA GPU and are running GeNN in CPU_ONLY mode, you can instead invoke this executable with

```
./generate_run --cpu-only test1
```

8.1.2 Windows

While GeNN can be used from within Visual Studio, in this example we will use a `cmd` window. Open a Visual Studio `cmd` window via Start: All Programs: Visual Studio: Tools: x86 Native Tools Command Prompt, and navigate to the `userproject\tools` directory. Then compile the additional tools and the `generate_run` executable for creating and running the project:

```
msbuild ...sln /t:generate_mbody1_runner /p:Configuration=Release
```

to generate an executable that you can invoke with

```
generate_run test1
```

or, if you don't have an NVIDIA GPU and are running GeNN in CPU_ONLY mode, you can instead invoke this executable with

```
generate_run --cpu-only test1
```

8.1.3 Visualising results

These steps will build and simulate a model of the locust olfactory system with default parameters of 100 projection neurons, 1000 Kenyon cells, 20 lateral horn interneurons and 100 output neurons in the mushroom body lobes. If the model isn't build in CPU_ONLY mode it will be simulated on an automatically chosen GPU.

The generate_run tool generates input patterns and writes them to file, compiles and runs the model using these files as inputs and finally output the resulting spiking activity. For more information of the options passed to this command see the *Insect olfaction model* section. The results of the simulation can be plotted with

```
python plot.py test1
```

The MBody1 example is already a highly integrated example that showcases many of the features of GeNN and how to program the user-side code for a GeNN application. More details in the *User Manual*.

8.2 How to use GeNN for New Projects

Creating and running projects in GeNN involves a few steps ranging from defining the fundamentals of the model, inputs to the model, details of the model like specific connectivity matrices or initial values, running the model, and analyzing or saving the data.

GeNN code is generally created by passing the C++ model file (see *below*) directly to the genn-buildmodel script. Another way to use GeNN is to create or modify a script or executable such as `userproject/MBody1_project/generate_run.cc` that wraps around the other programs that are used for each of the steps listed above. In more detail, the GeNN workflow consists of:

1. Either use external programs to generate connectivity and input files to be loaded into the user side code at runtime or generate these matrices directly inside the user side code.
2. Generating the model simulation code using `genn-buildmodel.sh` (On Linux or Mac) or `genn-buildmodel.bat` (on Windows). For example, inside the `generate_run` engine used by the MBody1_project, the following command is executed on Linux:

   ```
   genn-buildmodel.sh MBody1.cc
   ```

   or, if you don't have an NVIDIA GPU and are running GeNN in CPU_ONLY mode, the following command is executed:

   ```
   genn-buildmodel.sh -c MBody1.cc
   ```

   The `genn-buildmodel` script compiles the GeNN code generator in conjunction with the user-provided model description `model/MBody1.cc`. It then executes the GeNN code generator to generate the complete model simulation code for the model.
3. Provide a build script to compile the generated model simulation and the user side code into a simulator executable (in the case of the MBody1 example this consists the file `MBody1Sim.cc`). On Linux or Mac a suitable GNU makefile can be created by running:

```
genn-create-user-project.sh MBody1 MBody1Sim.cc
```

And on Windows an MSBuild project can be created by running:

```
genn-create-user-project.bat MBody1 MBody1Sim.cc
```

4. Compile the simulator executable by invoking GNU make on Linux or Mac:

```
make clean all
```

or MSbuild on Windows:

```
msbuild MBody1.sln /t:MBody1 /p:Configuration=Release
```

5. Finally, run the resulting stand-alone simulator executable. In the MBody1 example, this is called `MBody1` on Linux and `MBody1_Release.exe` on Windows.

8.3 Defining a New Model in GeNN

According to the work flow outlined above, there are several steps to be completed to define a neuronal network model.

1. The neuronal network of interest is defined in a model definition file, e.g. `Example1.cc`.
2. Within the the model definition file `Example1.cc`, the following tasks need to be completed:

   a) The GeNN file `modelSpec.h` needs to be included,

```
#include "modelSpec.h"
```

   b) The values for initial variables and parameters for neuron and synapse populations need to be defined, e.g.

```
NeuronModels::PoissonNew::ParamValues poissonParams(
  10.0);       // 0 - firing rate
```

   would define the (homogeneous) parameters for a population of Poisson neurons. The number of required parameters and their meaning is defined by the neuron or synapse type. Refer to the *User Manual* for details. We recommend, however, to use comments like in the above example to achieve maximal clarity of each parameter's meaning.

   If heterogeneous parameter values are required for a particular population of neurons (or synapses), they need to be defined as "variables" rather than parameters. See the *User Manual* for how to define new neuron (or synapse) types and the *Defining a new variable initialisation snippet* section for more information on initialising these variables to hetererogenous values.

   c) The actual network needs to be defined in the form of a function `modelDefinition`, i.e.

```
void modelDefinition(ModelSpec &model);
```

   The name `modelDefinition` and its parameter of type `ModelSpec&` are fixed and cannot be changed if GeNN is to recognize it as a model definition.

   d) Inside modelDefinition(), The time step `DT` needs to be defined, e.g.

```
model.setDT(0.1);
```

   All provided examples and pre-defined model elements in GeNN work with units of mV, ms, nF and muS. However, the choice of units is entirely left to the user if custom model elements are used.

   `MBody1.cc` shows a typical example of a model definition function. In its core it contains calls to *ModelSpec::addNeuronPopulation* and *ModelSpec::addSynapsePopulation* to build up the network. For a full range of options for defining a network, refer to the *User Manual*.

3. The programmer defines their own "user-side" modeling code similar to the code in `userproject/MBody1_project/model/MBody1Sim.cc`. In this code,

a) They manually define the connectivity matrices between neuron groups. Refer to the *Synaptic matrix types* section for the required format of connectivity matrices for dense or sparse connectivities.

b) They define input patterns (e.g. for Poisson neurons like in the MBody1 example) or individual initial values for neuron and / or synapse variables. The initial values given in the `modelDefinition` are automatically applied homogeneously to every individual neuron or synapse in each of the neuron or synapse groups.

c) They use `stepTime()` to run one time step on either the CPU or GPU depending on the options passed to genn-buildmodel.

d) They use functions like `copyStateFromDevice()` etc to transfer the results from GPU calculations to the main memory of the host computer for further processing.

e) They analyze the results. In the most simple case this could just be writing the relevant data to output files.

Previous | *Top* | *Next*

CHAPTER 9

Release Notes

CHAPTER 10

Release Notes for GeNN v4.0.2

This release fixes several small issues with the generation of binary wheels for Python:

10.1 Bug fixes:

1. There was a conflict between the versions of numpy used to build the wheels and the version required for the PyGeNN packages
2. Wheels were renamed to include the CUDA version which broke them.

CHAPTER 11

Release Notes for GeNN v4.0.1

This release fixes several small bugs found in GeNN 4.0.0 and implements some small features:

11.1 User Side Changes

1. Improved detection and handling of errors when specifying model parameters and values in PyGeNN.
2. SpineML simulator is now implemented as a library which can be used directly from user applications as well as from command line tool.

11.2 Bug fixes:

1. Fixed typo in `GeNNModel.push_var_to_device` function in PyGeNN.
2. Fixed broken support for Visual C++ 2013.
3. Fixed zero-copy mode.
4. Fixed typo in tutorial 2.

CHAPTER 12

Release Notes for GeNN v4.0.0

This release is the result of a second round of fairly major refactoring which we hope will make GeNN easier to use and allow it to be extended more easily in future. However, especially if you have been using GeNN 2.XX syntax, it breaks backward compatibility.

12.1 User Side Changes

1. Totally new build system - `make install` can be used to install GeNN to a system location on Linux and Mac and Windows projects work much better in the Visual Studio IDE.
2. Python interface now supports Windows and can be installed using binary 'wheels' (see *Python interface (PyGeNN)* for more details).
3. No need to call `initGeNN()` at start and `model.finalize()` at end of all models.
4. Initialisation system simplified - if you specify a value or initialiser for a variable or sparse connectivity, it will be initialised by your chosen backend. If you mark it as uninitialised, it is up to you to initialize it in user code between the calls to `initialize()` and `initializeSparse()` (where it will be copied to device).
5. `genn-create-user-project` helper scripts to create Makefiles or MSBuild projects for building user code
6. State variables can now be pushed and pulled individually using the `pull<var name><neuron or synapse name>FromDevice()` and `push<var name><neuron or synapse name>ToDevice()` functions.
7. Management of extra global parameter arrays has been somewhat automated (see *Extra Global Parameters* for more details).
8. `GENN_PREFERENCES` is no longer a namespace - it's a global struct so members need to be accessed with . rather than ::.
9. `NeuronGroup`, `SynapseGroup`, `CurrentSource` and `NNmodel` all previously exposed a lot of methods that the user wasn't *supposed* to call but could. These have now all been made protected and are exposed to GeNN internals using derived classes (`NeuronGroupInternal`, `SynapseGroupInternal`, `CurrentSourceInternal`, `ModelSpecInternal`) that make them public using `using` directives.

10. Auto-refractory behaviour was controlled using `GENN_PREFERENCES::autoRefractory`, this is now controlled on a per-neuron-model basis using the `SET_NEEDS_AUTO_REFRACTORY` macro.
11. The functions used for pushing and pulling have been unified somewhat this means that `copyStateToDevice` and `copyStateFromDevice` functions no longer copy spikes and `pus<neuron or synapse name>SpikesToDevice` and `pull<neuron or synapse name>SpikesFromDevice` no longer copy spike times or spike-like events.
12. Standard models of leaky-integrate-and-fire neuron (`NeuronModels::LIF`) and of exponentially shaped postsynaptic current (`PostsynapticModels::ExpCurr`) have been added.
13. When a model is built using the CUDA backend, the device it was built for is stored using it's PCI bus ID so it will always use the same device.

12.2 Deprecations

1. Yale-format sparse matrices are no longer supported.
2. GeNN 2.X syntax for implementing neuron and synapse models is no longer supported.
3. $(addtoinSyn) = X; $(updatelinsyn); idiom in weight update models has been replaced by function style `$(addToInSyn, X);`.

CHAPTER 13

Release Notes for GeNN v3.3.0

This release is intended as the last service release for GeNN 3.X.X. Fixes for serious bugs **may** be backported if requested but, otherwise, development will be switching to GeNN 4.

13.1 User Side Changes

1. Postsynaptic models can now have Extra Global Parameters.
2. Gamma distribution can now be sampled using `$(gennrand_gamma, a)`. This can be used to initialise variables using `InitVarSnippet::Gamma`.
3. Experimental Python interface - All features of GeNN are now exposed to Python through the *pygenn* module (see *Python interface (PyGeNN)* for more details).

13.2 Bug fixes:

1. Devices with Streaming Multiprocessor version 2.1 (compute capability 2.0) now work correctly in Windows.
2. Seeding of on-device RNGs now works correctly.
3. Improvements to accuracy of memory usage estimates provided by code generator.

CHAPTER 14

Release Notes for GeNN v3.2.0

This release extends the initialisation system introduced in 3.1.0 to support the initialisation of sparse synaptic connectivity, adds support for networks with more sophisticated models of synaptic plasticity and delay as well as including several other small features, optimisations and bug fixes for certain system configurations. This release supports GCC >= 4.9.1 on Linux, Visual Studio >= 2013 on Windows and recent versions of Clang on Mac OS X.

14.1 User Side Changes

1. Sparse synaptic connectivity can now be initialised using small *snippets* of code run either on GPU or CPU. This can save significant amounts of initialisation time for large models. See *Sparse connectivity initialisation* for more details.
2. New 'ragged matrix' data structure for representing sparse synaptic connections supports initialisation using new sparse synaptic connecivity initialisation system and enables future optimisations. See *Synaptic matrix types* for more details.
3. Added support for pre and postsynaptic state variables for weight update models to allow more efficient implementatation of trace based STDP rules. See *Defining a new weight update model* for more details.
4. Added support for devices with Compute Capability 7.0 (Volta) to block-size optimizer.
5. Added support for a new class of 'current source' model which allows non-synaptic input to be efficiently injected into neurons. See *Current source models* for more details.
6. Added support for heterogeneous dendritic delays. See *Defining a new weight update model* for more details.
7. Added support for (homogeneous) synaptic back propagation delays using `SynapseGroup::setBackPropDelaySteps`.
8. For long simulations, using single precision to represent simulation time does not work well. Added `NNmodel::setTimePrecision` to allow data type used to represent time to be set independently.

14.2 Optimisations

1. `GENN_PREFERENCES::mergePostsynapticModels` flag can be used to enable the merging together of postsynaptic models from a neuron population's incoming synapse populations - improves performance and saves memory.
2. On devices with compute capability > 3.5 GeNN now uses the read only cache to improve performance of postsynaptic learning kernel.

14.3 Bug fixes:

1. Fixed bug enabling support for CUDA 9.1 and 9.2 on Windows.
2. Fixed bug in SynDelay example where membrane voltage went to NaN.
3. Fixed bug in code generation of `SCALAR_MIN` and `SCALAR_MAX` values.
4. Fixed bug in substitution of trancendental functions with single-precision variants.
5. Fixed various issues involving using spike times with delayed synapse projections.

CHAPTER 15

Release Notes for GeNN v3.1.1

This release fixes several small bugs found in GeNN 3.1.0 and implements some small features:

15.1 User Side Changes

1. Added new synapse matrix types `SPARSE_GLOBALG_INDIVIDUAL_PSM`, `DENSE_GLOBALG_INDIVIDUAL_PSM` and `BITMASK_GLOBALG_INDIVIDUAL_PSM` to handle case where synapses with no individual state have a postsynaptic model with state variables e.g. an alpha synapse. See *Synaptic matrix types* for more details.

15.2 Bug fixes

1. Correctly handle aliases which refer to other aliases in SpineML models.
2. Fixed issues with presynaptically parallelised synapse populations where the postsynaptic population is small enough for input to be accumulated in shared memory.

CHAPTER 16

Release Notes for GeNN v3.1.0

This release builds on the changes made in 3.0.0 to further streamline the process of building models with GeNN and includes several bug fixes for certain system configurations.

16.1 User Side Changes

1. Support for simulating models described using the SpineML model description language with GeNN (see *SpineML and SpineCreator* for more details).
2. Neuron models can now sample from uniform, normal, exponential or log-normal distributions - these calls are translated to cuRAND when run on GPUs and calls to the C++11 `<random>` library when run on CPU. See *Defining your own neuron type* for more details.
3. Model state variables can now be initialised using small *snippets* of code run either on GPU or CPU. This can save significant amounts of initialisation time for large models. See *Defining a new variable initialisation snippet* for more details.
4. New MSBuild build system for Windows - makes developing user code from within Visual Studio much more streamlined. See *Debugging suggestions* for more details.

16.2 Bug fixes:

1. Workaround for bug found in Glibc 2.23 and 2.24 which causes poor performance on some 64-bit Linux systems (namely on Ubuntu 16.04 LTS).
2. Fixed bug encountered when using extra global variables in weight updates.

CHAPTER 17

Release Notes for GeNN v3.0.0

This release is the result of some fairly major refactoring of GeNN which we hope will make it more user-friendly and maintainable in the future.

17.1 User Side Changes

1. Entirely new syntax for defining models - hopefully terser and less error-prone (see updated documentation and examples for details).
2. Continuous integration testing using Jenkins - automated testing and code coverage calculation calculated automatically for Github pull requests etc.
3. Support for using Zero-copy memory for model variables. Especially on devices such as NVIDIA Jetson TX1 with no physical GPU memory this can significantly improve performance when recording data or injecting it to the simulation from external sensors.

CHAPTER 18

Release Notes for GeNN v2.2.3

This release includes minor new features and several bug fixes for certain system configurations.

18.1 User Side Changes

1. Transitioned feature tests to use Google Test framework.
2. Added support for CUDA shader model 6.X

18.2 Bug fixes:

1. Fixed problem using GeNN on systems running 32-bit Linux kernels on a 64-bit architecture (Nvidia Jetson modules running old software for example).
2. Fixed problem linking against CUDA on Mac OS X El Capitan due to SIP (System Integrity Protection).
3. Fixed problems with support code relating to its scope and usage in spike-like event threshold code.
4. Disabled use of C++ regular expressions on older versions of GCC.

CHAPTER 19

Release Notes for GeNN v2.2.2

This release includes minor new features and several bug fixes for certain system configurations.

19.1 User Side Changes

1. Added support for the new version (2.0) of the Brian simulation package for Python.
2. Added a mechanism for setting user-defined flags for the C++ compiler and NVCC compiler, via `GENN_PREFERENCES`.

19.2 Bug fixes:

1. Fixed a problem with `atomicAdd()` redefinitions on certain CUDA runtime versions and GPU configurations.
2. Fixed an incorrect bracket placement bug in code generation for certain models.
3. Fixed an incorrect neuron group indexing bug in the learning kernel, for certain models.
4. The dry-run compile phase now stores temporary files in the current directory, rather than the temp directory, solving issues on some systems.
5. The `LINK_FLAGS` and `INCLUDE_FLAGS` in the common windows makefile include 'make-file_commin_win.mk' are now appended to, rather than being overwritten, fixing issues with custom user makefiles on Windows.

CHAPTER 20

Release Notes for GeNN v2.2.1

This bugfix release fixes some critical bugs which occur on certain system configurations.

20.1 Bug fixes:

1. (important) Fixed a Windows-specific bug where the CL compiler terminates, incorrectly reporting that the nested scope limit has been exceeded, when a large number of device variables need to be initialised.
2. (important) Fixed a bug where, in certain circumstances, outdated generateALL objects are used by the Makefiles, rather than being cleaned and replaced by up-to-date ones.
3. (important) Fixed an 'atomicAdd' redeclared or missing bug, which happens on certain CUDA architectures when using the newest CUDA 8.0 RC toolkit.
4. (minor) The SynDelay example project now correctly reports spike indexes for the input group.

Please refer to the full documentation for further details, tutorials and complete code documentation.

CHAPTER 21

Release Notes for GeNN v2.2

This release includes minor new features, some core code improvements and several bug fixes on GeNN v2.1.

21.1 User Side Changes

1. GeNN now analyses automatically which parameters each kernel needs access to and these and only these are passed in the kernel argument list in addition to the global time t. These parameters can be a combination of extraGlobalNeuronKernelParameters and extraGlobalSynapseKernelParameters in either neuron or synapse kernel. In the unlikely case that users wish to call kernels directly, the correct call can be found in the `stepTimeGPU()` function.

   Reflecting these changes, the predefined Poisson neurons now simply have two extraGlobalNeuronParameter `rates` and `offset` which replace the previous custom pointer to the array of input rates and integer offset to indicate the current input pattern. These extraGlobalNeuronKernelParameters are passed to the neuron kernel automatically, but the rates themselves within the array are of course not updated automatically (this is exactly as before with the specifically generated kernel arguments for Poisson neurons).

   The concept of "directInput" has been removed. Users can easily achieve the same functionality by adding an additional variable (if there are individual inputs to neurons), an extraGlobalNeuronParameter (if the input is homogeneous but time dependent) or, obviously, a simple parameter if it's homogeneous and constant. The global time variable "t" is now provided by GeNN; please make sure that you are not duplicating its definition or shadowing it. This could have severe consequences for simulation correctness (e.g. time not advancing in cases of over-shadowing).

2. We introduced the namespace GENN_PREFERENCES which contains variables that determine the behaviour of GeNN.

3. We introduced a new code snippet called "supportCode" for neuron models, weightupdate models and postsynaptic models. This code snippet is intended to contain user-defined functions that are used from the other code snippets. We advise where possible to define the support code functions with the CUDA keywords "__host__ __device__" so that they are available for both GPU and CPU version. Alternatively one can define separate versions for **host** and **device** in the snippet. The snippets are automatically made available to the relevant code parts. This is regulated through namespaces so that name clashes between different models do not

matter. An exception are hash defines. They can in principle be used in the supportCode snippet but need to be protected specifically using ifndef. For example

```
#ifndef clip(x)
#define clip(x) x > 10.0? 10.0 : x
#endif
```

If there are conflicting definitions for hash defines, the one that appears first in the GeNN generated code will then prevail.

4. The new convenience macros spikeCount_XX and spike_XX where "XX" is the name of the neuron group are now also available for events: spikeEventCount_XX and spikeEvent_XX. They access the values for the current time step even if there are synaptic delays and spikes events are stored in circular queues.

5. The old buildmodel.[sh|bat] scripts have been superseded by new genn-buildmodel.[sh|bat] scripts. These scripts accept UNIX style option switches, allow both relative and absolute model file paths, and allow the user to specify the directory in which all output files are placed (-o <path>). Debug (-d), CPU-only (-c) and show help (-h) are also defined.

6. We have introduced a CPU-only "-c" genn-buildmodel switch, which, if it's defined, will generate a GeNN version that is completely independent from CUDA and hence can be used on computers without CUDA installation or CUDA enabled hardware. Obviously, this then can also only run on CPU. CPU only mode can either be switched on by defining CPU_ONLY in the model description file or by passing appropriate parameters during the build, in particular

```
genn-buildmodel.[sh|bat] modelfile -c
make release CPU_ONLY=1
```

7. The new genn-buildmodel "-o" switch allows the user to specify the output directory for all generated files - the default is the current directory. For example, a user project could be in '/home/genn_project', whilst the GeNN directory could be '/usr/local/genn'. The GeNN directory is kept clean, unless the user decides to build the sample projects inside of it without copying them elsewhere. This allows the deployment of GeNN to a read-only directory, like '/usr/local' or 'C:Program Files'. It also allows multiple users - i.e. on a compute cluster - to use GeNN simultaneously, without overwriting each other's code-generation files, etcetera.

8. The ARM architecture is now supported - e.g. the NVIDIA Jetson development platform.

9. The NVIDIA CUDA SM_5* (Maxwell) architecture is now supported.

10. An error is now thrown when the user tries to use double precision floating-point numbers on devices with architecture older than SM_13, since these devices do not support double precision.

11. All GeNN helper functions and classes, such as `toString()` and `NNmodel`, are defined in the header files at `genn/lib/include/`, for example `stringUtils.h` and `modelSpec.h`, which should be individually included before the functions and classes may be used. The functions and classes are actually implementated in the static library `genn\lib\lib\genn.lib` (Windows) or `genn/lib/lib/libgenn.a` (Mac, Linux), which must be linked into the final executable if any GeNN functions or classes are used.

12. In the `modelDefinition()` file, only the header file `modelSpec.h` should be included - i.e. not the source file `modelSpec.cc`. This is because the declaration and definition of `NNmodel`, and associated functions, has been separated into `modelSpec.h` and `modelSpec.cc`, respectively. This is to enable NNmodel code to be precompiled separately. Henceforth, only the header file `modelSpec.h` should be included in model definition files!

13. In the `modelDefinition()` file, DT is now preferrably defined using `model.setDT(<val>);`, rather than # `define DT <val>`, in order to prevent problems with DT macro redefinition. For backward-compatibility reasons, the old # `define DT <val>` method may still be used, however users are advised to adopt the new method.

14. In preparation for multi-GPU support in GeNN, we have separated out the compilation of generated code from user-side code. This will eventually allow us to optimise and compile different parts of the model with different

CUDA flags, depending on the CUDA device chosen to execute that particular part of the model. As such, we have had to use a header file `definitions.h` as the generated code interface, rather than the `runner.cc` file. In practice, this means that user-side code should include `myModel_CODE/definitions.h`, rather than `myModel_CODE/runner.cc`. Including `runner.cc` will likely result in pages of linking errors at best!

21.2 Developer Side Changes

1. Blocksize optimization and device choice now obtain the ptxas information on memory usage from a CUDA driver API call rather than from parsing ptxas output of the nvcc compiler. This adds robustness to any change in the syntax of the compiler output.
2. The information about device choice is now stored in variables in the namespace `GENN_PREFERENCES`. This includes `chooseDevice`, `optimiseBlockSize`, `optimizeCode`, `debugCode`, `showPtxInfo`, `defaultDevice`. `asGoodAsZero` has also been moved into this namespace.
3. We have also introduced the namespace GENN_FLAGS that contains unsigned int variables that attach names to numeric flags that can be used within GeNN.
4. The definitions of all generated variables and functions such as pullXXXStateFromDevice etc, are now generated into definitions.h. This is useful where one wants to compile separate object files that cannot all include the full definitions in e.g. "runnerGPU.cc". One example where this is useful is the brian2genn interface.
5. A number of feature tests have been added that can be found in the `featureTests` directory. They can be run with the respective `runTests.sh` scripts. The `cleanTests.sh` scripts can be used to remove all generated code after testing.

21.3 Improvements

1. Improved method of obtaining ptxas compiler information on register and shared memory usage and an improved algorithm for estimating shared memory usage requirements for different block sizes.
2. Replaced pageable CPU-side memory with page-locked memory. This can significantly speed up simulations in which a lot of data is regularly copied to and from a CUDA device.
3. GeNN library objects and the main generateALL binary objects are now compiled separately, and only when a change has been made to an object's source, rather than recompiling all software for a minor change in a single source file. This should speed up compilation in some instances.

21.4 Bug fixes:

1. Fixed a minor bug with delayed synapses, where delaySlot is declared but not referenced.
2. We fixed a bug where on rare occasions a synchronisation problem occurred in sparse synapse populations.
3. We fixed a bug where the combined spike event condition from several synapse populations was not assembled correctly in the code generation phase (the parameter values of the first synapse population over-rode the values of all other populations in the combined condition).

Please refer to the full documentation for further details, tutorials and complete code documentation.

CHAPTER 22

Release Notes for GeNN v2.1

This release includes some new features and several bug fixes on GeNN v2.0.

22.1 User Side Changes

1. Block size debugging flag and the asGoodAsZero variables are moved into include/global.h.
2. NGRADSYNAPSES dynamics have changed (See Bug fix #4) and this change is applied to the example projects. If you are using this synapse model, you may want to consider changing model parameters.
3. The delay slots are now such that NO_DELAY is 0 delay slots (previously 1) and 1 means an actual delay of 1 time step.
4. The convenience function convertProbabilityToRandomNumberThreshold(float *, uint64_t *, int) was changed so that it actually converts firing probability/timestep into a threshold value for the GeNN random number generator (as its name always suggested). The previous functionality of converting a *rate* in kHz into a firing threshold number for the GeNN random number generator is now provided with the name convertRateToRandomNumberThreshold(float *, uint64_t *, int)
5. Every model definition function `modelDefinition()` now needs to end with calling `NNmodel::finalize()` for the defined network model. This will lock down the model and prevent any further changes to it by the supported methods. It also triggers necessary analysis of the model structure that should only be performed once. If the `finalize()` function is not called, GeNN will issue an error and exit before code generation.
6. To be more consistent in function naming the `pull\<SYNAPSENAME\>FromDevice` and `push\<SYNAPSENAME\>ToDevice` have been renamed to `pull\<SYNAPSENAME\>StateFromDevice` and `push\<SYNAPSENAME\>StateToDevice`. The old versions are still supported through macro definitions to make the transition easier.
7. New convenience macros are now provided to access the current spike numbers and identities of neurons that spiked. These are called spikeCount_XX and spike_XX where "XX" is the name of the neuron group. They access the values for the current time step even if there are synaptic delays and spikes are stored in circular queues.

8. There is now a pre-defined neuron type "SPIKECOURCE" which is empty and can be used to define PyNN style spike source arrays.
9. The macros FLOAT and DOUBLE were replaced with GENN_FLOAT and GENN_DOUBLE due to name clashes with typedefs in Windows that define FLOAT and DOUBLE.

22.2 Developer Side Changes

1. We introduced a file definitions.h, which is generated and filled with useful macros such as spkQuePtrShift which tells users where in the circular spike queue their spikes start.

22.3 Improvements

1. Improved debugging information for block size optimisation and device choice.
2. Changed the device selection logic so that device occupancy has larger priority than device capability version.
3. A new HH model called TRAUBMILES_PSTEP where one can set the number of inner loops as a parameter is introduced. It uses the TRAUBMILES_SAFE method.
4. An alternative method is added for the insect olfaction model in order to fix the number of connections to a maximum of 10K in order to avoid negative conductance tails.
5. We introduced a preprocessor define directive for an "int_" function that translates floating points to integers.

22.4 Bug fixes:

1. AtomicAdd replacement for old GPUs were used by mistake if the model runs in double precision.
2. Timing of individual kernels is fixed and improved.
3. More careful setting of maximum number of connections in sparse connectivity, covering mixed dense/sparse network scenarios.
4. NGRADSYNAPSES was not scaling correctly with varying time step.
5. Fixed a bug where learning kernel with sparse connectivity was going out of range in an array.
6. Fixed synapse kernel name substitutions where the "dd_" prefix was omitted by mistake.

Please refer to the full documentation for further details, tutorials and complete code documentation.

CHAPTER 23

Release Notes for GeNN v2.0

Version 2.0 of GeNN comes with a lot of improvements and added features, some of which have necessitated some changes to the structure of parameter arrays among others.

23.1 User Side Changes

1. Users are now required to call `initGeNN()` in the model definition function before adding any populations to the neuronal network model.
2. glbscnt is now call glbSpkCnt for consistency with glbSpkEvntCnt.
3. There is no longer a privileged parameter `Epre`. Spike type events are now defined by a code string `spkEvntThreshold`, the same way proper spikes are. The only difference is that Spike type events are specific to a synapse type rather than a neuron type.
4. The function setSynapseG has been deprecated. In a `GLOBALG` scenario, the variables of a synapse group are set to the initial values provided in the `modeldefinition` function.
5. Due to the split of synaptic models into weightUpdateModel and postSynModel, the parameter arrays used during model definition need to be carefully split as well so that each side gets the right parameters. For example, previously

   ```
   float myPNKC_p[3]=
   0.0,           // 0 - Erev: Reversal potential
   -20.0,         // 1 - Epre: Presynaptic threshold potential
   1.0            // 2 - tau_S: decay time constant for S [ms]
   ;
   ```

   would define the parameter array of three parameters, `Erev`, `Epre`, and `tau_S` for a synapse of type `NSYNAPSE`. This now needs to be "split" into

   ```
   float *myPNKC_p= NULL;
   float postExpPNKC[2]=
     1.0,            // 0 - tau_S: decay time constant for S [ms]
     0.0         // 1 - Erev: Reversal potential
   ```

;

i.e. parameters Erev and tau_S are moved to the post-synaptic model and its parameter array of two parameters. Epre is discontinued as a parameter for NSYNAPSE. As a consequence the weightupdate model of NSYNAPSE has no parameters and one can pass NULL for the parameter array in addSynapsePopulation. The correct parameter lists for all defined neuron and synapse model types are listed in the User Manual. If the parameters are not redefined appropriately this will lead to uncontrolled behaviour of models and likely to segmentation faults and crashes.

6. Advanced users can now define variables as type scalar when introducing new neuron or synapse types. This will at the code generation stage be translated to the model's floating point type (ftype), float or double. This works for defining variables as well as in all code snippets. Users can also use the expressions SCALAR_MAX and SCALAR_MIN for FLT_MIN, FLT_MAX, DBL_MIN and DBL_MAX, respectively. Corresponding definitions of scalar, SCALAR_MIN and SCALAR_MAX are also available for user-side code whenever the code-generated file runner.cc has been included.

7. The example projects have been re-organized so that wrapper scripts of the generate_run type are now all located together with the models they run instead of in a common tools directory. Generally the structure now is that each example project contains the wrapper script generate_run and a model subdirectory which contains the model description file and the user side code complete with Makefiles for Unix and Windows operating systems. The generated code will be deposited in the model subdirectory in its own modelname_CODE folder. Simulation results will always be deposited in a new sub-folder of the main project directory.

8. The addSynapsePopulation(...) function has now more mandatory parameters relating to the introduction of separate weightupdate models (pre-synaptic models) and postynaptic models. The correct syntax for the addSynapsePopulation(...) can be found with detailed explanations in teh User Manual.

9. We have introduced a simple performance profiling method that users can employ to get an overview over the differential use of time by different kernels. To enable the timers in GeNN generated code, one needs to declare

   ```
   networkmodel.setTiming(TRUE);
   ```

   This will make available and operate GPU-side cudeEvent based timers whose cumulative value can be found in the double precision variables neuron_tme, synapse_tme and learning_tme. They measure the accumulated time that has been spent calculating the neuron kernel, synapse kernel and learning kernel, respectively. CPU-side timers for the simulation functions are also available and their cumulative values can be obtained through

   ```
   float x= sdkGetTimerValue(&neuron_timer);
   float y= sdkGetTimerValue(&synapse_timer);
   float z= sdkGetTimerValue(&learning_timer);
   ```

   The *Insect olfaction model* example shows how these can be used in the user-side code. To enable timing profiling in this example, simply enable it for GeNN:

   ```
   model.setTiming(TRUE);
   ```

   in MBody1.cc 's modelDefinition function and define the macro TIMING in classol_sim.h

   ```
   #define TIMING
   ```

   This will have the effect that timing information is output into OUTNAME_output/OUTNAME.timingprofile.

23.2 Developer Side Changes

1. allocateSparseArrays() has been changed to take the number of connections, connN, as an argument rather than expecting it to have been set in the Connetion struct before the function is called as was the arrangement previously.

2. For the case of sparse connectivity, there is now a reverse mapping implemented with revers index arrays and a remap array that points to the original positions of variable values in teh forward array. By this mechanism, revers lookups from post to pre synaptic indices are possible but value changes in the sparse array values do only need to be done once.
3. SpkEvnt code is no longer generated whenever it is not actually used. That is also true on a somewhat finer granularity where variable queues for synapse delays are only maintained if the corresponding variables are used in synaptic code. True spikes on the other hand are always detected in case the user is interested in them.

Please refer to the full documentation for further details, tutorials and complete code documentation.

Previous | *Top* | *Next*

CHAPTER 24

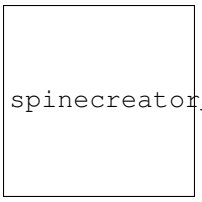
SpineML and SpineCreator

GeNN now supports simulating models built using SpineML and includes scripts to fully integrate it with the SpineCreator graphical editor on Linux, Mac and Windows. After installing GeNN using the instructions in *Installation*, build SpineCreator for your platform.

From SpineCreator, select Edit->Settings->Simulators and add a new simulator using the following settings (replacing "/home/j/jk/jk421/genn" with the GeNN installation directory on your own system):

If you would like SpineCreator to use GeNN in CPU only mode, add an environment variable called "GENN_SPINEML_CPU_ONLY".

The best way to get started using SpineML with GeNN is to experiment with some example models. A number are available here although the "Striatal model" uses features not currently supported by GeNN and the two "Brette Benchmark" models use a legacy syntax no longer supported by SpineCreator (or GeNN). Once you have loaded a model, click "Expts" from the menu on the left hand side of SpineCreator, choose the experiment you would like to run and then select your newly created GeNN simulator in the "Setup Simulator" panel:



Now click “Run experiment” and, after a short time, the results of your GeNN simulation will be available for plotting by clicking the “Graphs” option in the menu on the left hand side of SpineCreator. *Previous* | *Top* | *Next*

CHAPTER 25

Tutorial 1

In this tutorial we will go through step by step instructions how to create and run your first GeNN simulation from scratch.

25.1 The Model Definition

In this tutorial we will use a pre-defined Hodgkin-Huxley neuron model (*NeuronModels::TraubMiles*) and create a simulation consisting of ten such neurons without any synaptic connections. We will run this simulation on a GPU and save the results - firstly to stdout and then to file.

The first step is to write a model definition function in a model definition file. Create a new directory and, within that, create a new empty file called `tenHHModel.cc` using your favourite text editor, e.g.

```
>> emacs tenHHModel.cc &
```

The ">>" in the example code snippets refers to a shell prompt in a unix shell, do not enter them as part of your shell commands.

The model definition file contains the definition of the network model we want to simulate. First, we need to include the GeNN model specification code `modelSpec.h`. Then the model definition takes the form of a function named `modelDefinition` that takes one argument, passed by reference, of type `ModelSpec`. Type in your `tenHHModel.cc` file:

```
// Model definintion file tenHHModel.cc

#include "modelSpec.h"

void modelDefinition(ModelSpec &model)

    // definition of tenHHModel
```

Two standard elements to the ‘modelDefinition function are setting the simulation step size and setting the name of the model:

```
model.setDT(0.1);
model.setName("tenHHModel");
```

With this we have fixed the integration time step to `0.1` in the usual time units. The typical units in GeNN are `ms`, `mV`, `nF`, and `S`. Therefore, this defines `DT= 0.1 ms`.

Making the actual model definition makes use of the *ModelSpec::addNeuronPopulation* and *ModelSpec::addSynapsePopulation* member functions of the *ModelSpec* object. The arguments to a call to *ModelSpec::addNeuronPopulation* are

- `NeuronModel` : template parameter specifying the neuron model class to use
- `const std::string &name` : the name of the population
- `unsigned int size` : The number of neurons in the population
- `const NeuronModel::ParamValues &paramValues` : Parameter values for the neurons in the population
- `const NeuronModel::VarValues &varInitialisers` : Initial values or initialisation snippets for variables of this neuron type

We first create the parameter and initial variable arrays,

```
// definition of tenHHModel
NeuronModels::TraubMiles::ParamValues p(
    7.15,       // 0 - gNa: Na conductance in muS
    50.0,       // 1 - ENa: Na equi potential in mV
    1.43,       // 2 - gK: K conductance in muS
    -95.0,      // 3 - EK: K equi potential in mV
    0.02672,    // 4 - gl: leak conductance in muS
    -63.563,    // 5 - El: leak equi potential in mV
    0.143);     // 6 - Cmem: membr. capacity density in nF

NeuronModels::TraubMiles::VarValues ini(
    -60.0,      // 0 - membrane potential V
    0.0529324,  // 1 - prob. for Na channel activation m
    0.3176767,  // 2 - prob. for not Na channel blocking h
    0.5961207); // 3 - prob. for K channel activation n
```

The comments are obviously only for clarity, they can in principle be omitted. To avoid any confusion about the meaning of parameters and variables, however, we recommend strongly to always include comments of this type.

Having defined the parameter values and initial values we can now create the neuron population,

```
model.addNeuronPopulation<NeuronModels::TraubMiles>("Pop1", 10, p, ini);
```

This completes the model definition in this example. The complete `tenHHModel.cc` file now should look like this:

```
// Model definintion file tenHHModel.cc

#include "modelSpec.h"

void modelDefinition(ModelSpec &model)

    // definition of tenHHModel
    model.setDT(0.1);
    model.setName("tenHHModel");

    NeuronModels::TraubMiles::ParamValues p(
        7.15,       // 0 - gNa: Na conductance in muS
```

```
        50.0,       // 1 - ENa: Na equi potential in mV
        1.43,       // 2 - gK: K conductance in muS
        -95.0,      // 3 - EK: K equi potential in mV
        0.02672,    // 4 - gl: leak conductance in muS
        -63.563,    // 5 - El: leak equi potential in mV
        0.143);     // 6 - Cmem: membr. capacity density in nF

    NeuronModels::TraubMiles::VarValues ini(
        -60.0,      // 0 - membrane potential V
        0.0529324,  // 1 - prob. for Na channel activation m
        0.3176767,  // 2 - prob. for not Na channel blocking h
        0.5961207); // 3 - prob. for K channel activation n

    model.addNeuronPopulation<NeuronModels::TraubMiles>("Pop1", 10, p, ini);
```

This model definition suffices to generate code for simulating the ten Hodgkin-Huxley neurons on the a GPU or CPU. The second part of a GeNN simulation is the user code that sets up the simulation, does the data handling for input and output and generally defines the numerical experiment to be run.

25.2 Building the model

To use GeNN to build your model description into simulation code, use a terminal to navigate to the directory containing your `tenHHModel.cc` file and, on Linux or Mac, type:

```
>> genn-buildmodel.sh tenHHModel.cc
```

Alternatively, on Windows, type:

```
>> genn-buildmodel.bat tenHHModel.cc
```

If you don't have an NVIDIA GPU and are running GeNN in CPU_ONLY mode, you can invoke `genn-buildmodel` with a `-c` option so, on Linux or Mac:

```
>> genn-buildmodel.sh -c tenHHModel.cc
```

or on Windows:

```
>> genn-buildmodel.bat -c tenHHModel.cc
```

If GeNN has been added to your path and `CUDA_PATH` is correctly configured, you should see some compile output ending in `Model build complete ...`.

25.3 User Code

GeNN will now have generated the code to simulate the model for one timestep using a function `stepTime()`. To make use of this code, we need to define a minimal C/C++ main function. For the purposes of this tutorial we will initially simply run the model for one simulated second and record the final neuron variables into a file. Open a new empty file `tenHHSimulation.cc` in an editor and type

```
// tenHHModel simulation code
#include "tenHHModel_CODE/definitions.h"

int main()

    allocateMem();
```

```
    initialize();
    return 0;
```

This boiler plate code includes the header file for the generated code `definitions.h` in the subdirectory `tenHHModel_CODE` where GeNN deposits all generated code (this corresponds to the name passed to the `ModelSpec::setName` function). Calling `allocateMem()` allocates the memory structures for all neuron variables and `initialize()` launches a GPU kernel which initialise all state variables to their initial values. Now we can use the generated code to integrate the neuron equations provided by GeNN for 1000ms. To do so, we add after `initialize();` The `t` variable is provided by GeNN to keep track of the current simulation time in milliseconds.

```
while (t < 1000.0f)
    stepTime();
```

and we need to copy the result back to the host before outputting it to stdout (this will do nothing if you are running the model on a CPU),

```
pullPop1StateFromDevice();
for (int j= 0; j < 10; j++)
    std::cout << VPop1[j] << " ";
    std::cout << mPop1[j] << " ";
    std::cout << hPop1[j] << " ";
    std::cout << nPop1[j] << std::endl;
```

`pullPop1StateFromDevice()` copies all relevant state variables of the `Pop1` neuron group from the GPU to the CPU main memory. Then we can output the results to stdout by looping through all 10 neurons and outputting the state variables VPop1, mPop1, hPop1, nPop1. The naming convention for variables in GeNN is the variable name defined by the neuron type, here TraubMiles defining V, m, h, and n, followed by the population name, here `Pop1`.

This completes the user code. The complete `tenHHSimulation.cc` file should now look like

```
// tenHHModel simulation code
#include "tenHHModel_CODE/definitions.h"

int main()

    allocateMem();
    initialize();

    while (t < 1000.0f)
        stepTime();

    pullPop1StateFromDevice();

    for (int j= 0; j < 10; j++)
        std::cout << VPop1[j] << " ";
        std::cout << mPop1[j] << " ";
        std::cout << hPop1[j] << " ";
        std::cout << nPop1[j] << std::endl;

    return 0;
```

25.4 Building the simulator (Linux or Mac)

On Linux and Mac, GeNN simulations are typically built using a simple Makefile which can be generated with the following command:

```
genn-create-user-project.sh tennHHModel tenHHSimulation.cc
```

This defines that the model is named tennHHModel and the simulation code is given in the file `tenHHSimulation.cc` that we completed above. Now type

```
make
```

25.5 Building the simulator (Windows)

So that projects can be easily debugged within the Visual Studio IDE (see section *Debugging suggestions* for more details), Windows projects are built using an MSBuild script typically with the same title as the final executable. A suitable solution and project can be generated automatically with the following command:

```
genn-create-user-project.bat tennHHModel tenHHSimulation.cc
```

his defines that the model is named tennHHModel and the simulation code is given in the file `tenHHSimulation.cc` that we completed above. Now type

```
msbuild tennHHModel.sln /p:Configuration=Release /t:tennHHModel
```

25.6 Running the Simulation

You can now execute your newly-built simulator on Linux or Mac with

```
./tennHHModel
```

Or on Windows with

```
tennHHModel_Release
```

The output you obtain should look like

```
-63.7838 0.0350042 0.336314 0.563243
-63.7838 0.0350042 0.336314 0.563243
-63.7838 0.0350042 0.336314 0.563243
-63.7838 0.0350042 0.336314 0.563243
-63.7838 0.0350042 0.336314 0.563243
-63.7838 0.0350042 0.336314 0.563243
-63.7838 0.0350042 0.336314 0.563243
-63.7838 0.0350042 0.336314 0.563243
-63.7838 0.0350042 0.336314 0.563243
-63.7838 0.0350042 0.336314 0.563243
```

25.7 Reading

This is not particularly interesting as we are just observing the final value of the membrane potentials. To see what is going on in the meantime, we need to copy intermediate values from the device and save them into a file. This can be done in many ways but one sensible way of doing this is to replace the calls to `stepTime` in `tenHHSimulation.cc` with something like this:

```
std::ofstream os("tenHH_output.V.dat");
while (t < 1000.0f)
    stepTime();

    pullVPop1FromDevice();

    os << t << " ";
    for (int j= 0; j < 10; j++)
        os << VPop1[j] << " ";

    os << std::endl;

os.close();
```

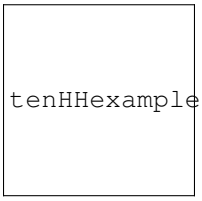
t is a global variable updated by the GeNN code to keep track of elapsed simulation time in ms.

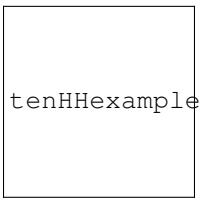
we switched from using `pullPop1StateFromDevice()` to `pullVPop1FromDevice()` as we are now only interested in the membrane voltage of the neuron.

You will also need to add:

```
#include <fstream>
```

to the top of tenHHSimulation.cc. After building the model; and building and running the simulator as described above there should be a file `tenHH_output.V.dat` in the same directory. If you plot column one (time) against the subsequent 10 columns (voltage of the 10 neurons), you should observe dynamics like this:





However so far, the neurons are not connected and do not receive input. As the *NeuronModels::TraubMiles* model is silent in such conditions, the membrane voltages of the 10 neurons will simply drift from the -60mV they were initialised at to their resting potential.

Previous | *Top* | *Next*

CHAPTER 26

Tutorial 2

In this tutorial we will learn to add synapsePopulations to connect neurons in neuron groups to each other with synaptic models. As an example we will connect the ten Hodgkin-Huxley neurons from tutorial 1 in a ring of excitatory synapses.

First, copy the files from Tutorial 1 into a new directory and rename the `tenHHModel.cc` to `tenHHRingModel.cc` and `tenHHSimulation.cc` to `tenHHRingSimulation.cc`, e.g. on Linux or Mac:

```
>> cp -r tenHH_project tenHHRing_project
>> cd tenHHRing_project
>> mv tenHHModel.cc tenHHRingModel.cc
>> mv tenHHSimulation.cc tenHHRingSimulation.cc
```

Finally, to reduce confusion we should rename the model itself. Open `tenHHRingModel.cc`, change the model name inside,

```
model.setName("tenHHRing");
```

26.1 Defining the Detailed Synaptic Connections

We want to connect our ten neurons into a ring where each neuron connects to its neighbours. In order to initialise this connectivity we need to add a sparse connectivity initialisation snippet at the top of `tenHHRingModel.cc`:

```
class Ring : public InitSparseConnectivitySnippet::Base

public:
    DECLARE_SNIPPET(Ring, 0);
    SET_ROW_BUILD_CODE(
        "$(addSynapse, ($(id_pre) + 1) % $(num_post));"
        "$(endRow);");
    SET_MAX_ROW_LENGTH(1);
;
IMPLEMENT_SNIPPET(Ring);
```

The SET_ROW_BUILD_CODE code string will be called to generate each row of the synaptic matrix (connections coming from a single presynaptic neuron) and, in this case, each row consists of a single synapses from the presynaptic neuron $(id_pre) to $(id_pre) + 1 (the modulus operator is used to ensure that the final connection between neuron 9 and 0 is made correctly). In order to allow GeNN to better optimise the generated code we also provide a maximum row length. In this case each row always contains only one synapse but, when more complex connectivity is used, the number of neurons in the pre and postsynaptic population as well as any parameters used to configure the snippet can be accessed from this function. When defining GeNN code strings, the $(VariableName) syntax is used to refer to variables provided by GeNN and the $(FunctionName, Parameter1,...) syntax is used to call functions provided by GeNN.

26.2 Adding Synaptic connections

Now we need additional initial values and parameters for the synapse and post-synaptic models. We will use the standard *WeightUpdateModels::StaticPulse* weight update model and *PostsynapticModels::ExpCond* post-synaptic model. They need the following initial variables and parameters:

```
WeightUpdateModels::StaticPulse::VarValues s_ini(
    -0.2); // 0 - g: the synaptic conductance value

PostsynapticModels::ExpCond::ParamValues ps_p(
    1.0,    // 0 - tau_S: decay time constant for S [ms]
    -80.0); // 1 - Erev: Reversal potential
```

the *WeightUpdateModels::StaticPulse* weight update model has no parameters and the *PostsynapticModels::ExpCond* post-synaptic model has no state variables.

We can then add a synapse population at the end of the modelDefinition(...) function,

```
model.addSynapsePopulation<WeightUpdateModels::StaticPulse,␣
↪PostsynapticModels::ExpCond>(
    "Pop1self", SynapseMatrixType::SPARSE_GLOBALG, 10,
    "Pop1", "Pop1",
    , s_ini,
    ps_p, ,
    initConnectivity<Ring>());
```

The addSynapsePopulation parameters are

- WeightUpdateModel: template parameter specifying the type of weight update model (derived from *WeightUpdateModels::Base*).
- PostsynapticModel: template parameter specifying the type of postsynaptic model (derived from *PostsynapticModels::Base*).
- name string containing unique name of synapse population.
- mtype how the synaptic matrix associated with this synapse population should be represented. Here SynapseMatrixType::SPARSE_GLOBALG means that there will be sparse connectivity and each connection will have the same weight (-0.2 as specified previously).
- delayStep integer specifying number of timesteps of propagation delay that spikes travelling through this synapses population should incur (or NO_DELAY for none)
- src string specifying name of presynaptic (source) population
- trg string specifying name of postsynaptic (target) population
- weightParamValues parameters for weight update model wrapped in WeightUpdateModel::ParamValues object.

- weightVarInitialisers initial values or initialisation snippets for the weight update model's state variables wrapped in a WeightUpdateModel::VarValues object.
- postsynapticParamValues parameters for postsynaptic model wrapped in PostsynapticModel::ParamValues object.
- postsynapticVarInitialisers initial values or initialisation snippets for the postsynaptic model wrapped in PostsynapticModel::VarValues object.
- connectivityInitialiser snippet and any paramaters (in this case there are none) used to initialise the synapse population's sparse connectivity.

Adding the addSynapsePopulation command to the model definition informs GeNN that there will be synapses between the named neuron populations, here between population `Pop1` and itself. At this point our model definition file `tenHHRingModel.cc` should look like this

```
// Model definition file tenHHRing.cc
#include "modelSpec.h"

class Ring : public InitSparseConnectivitySnippet::Base

public:
    DECLARE_SNIPPET(Ring, 0);
    SET_ROW_BUILD_CODE(
        "$(addSynapse, ($(id_pre) + 1) % $(num_post));"
        "$(endRow);");
    SET_MAX_ROW_LENGTH(1);
;
IMPLEMENT_SNIPPET(Ring);

void modelDefinition(ModelSpec &model)

    // definition of tenHHRing
    model.setDT(0.1);
    model.setName("tenHHRing");

    NeuronModels::TraubMiles::ParamValues p(
        7.15,       // 0 - gNa: Na conductance in muS
        50.0,       // 1 - ENa: Na equi potential in mV
        1.43,       // 2 - gK: K conductance in muS
        -95.0,      // 3 - EK: K equi potential in mV
        0.02672,    // 4 - gl: leak conductance in muS
        -63.563,    // 5 - El: leak equi potential in mV
        0.143);     // 6 - Cmem: membr. capacity density in nF

    NeuronModels::TraubMiles::VarValues ini(
        -60.0,         // 0 - membrane potential V
        0.0529324,     // 1 - prob. for Na channel activation m
        0.3176767,     // 2 - prob. for not Na channel blocking h
        0.5961207);    // 3 - prob. for K channel activation n

    model.addNeuronPopulation<NeuronModels::TraubMiles>("Pop1", 10, p, ini);

    WeightUpdateModels::StaticPulse::VarValues s_ini(
         -0.2); // 0 - g: the synaptic conductance value
```

```
    PostsynapticModels::ExpCond::ParamValues ps_p(
        1.0,    // 0 - tau_S: decay time constant for S [ms]
        -80.0); // 1 - Erev: Reversal potential

    model.addSynapsePopulation<WeightUpdateModels::StaticPulse,␣
↪PostsynapticModels::ExpCond>(
        "Pop1self", SynapseMatrixType::SPARSE_GLOBALG, 100,
        "Pop1", "Pop1",
        , s_ini,
        ps_p, ,
        initConnectivity<Ring>());
```

We can now build our new model:

```
>> genn-buildmodel.sh tenHHRingModel.cc
```

Again, if you don't have an NVIDIA GPU and are running GeNN in CPU_ONLY mode, you can instead build with the `-c` option as described in *Tutorial 1*.

Now we can open the `tenHHRingSimulation.cc` file and update the file name of the model includes to match the name we set previously:

```
// tenHHRingModel simulation code
#include "tenHHRing_CODE/definitions.h"
```

Additionally, we need to add a call to a second initialisation function to `main()` after we call `initialize()`:

```
initializeSparse();
```

This initializes any variables associated with the sparse connectivity we have added (and will also copy any manually initialised variables to the GPU). Then, after using the `genn-create-user-project` tool to create a new project with a model name of `tenHHRing` and using `tenHHRingSimulation.cc` rather than `tenHHSimulation.cc`, we can build and run our new simulator in the same way we did in *Tutorial 1*. However, even after all our hard work, if we plot the content of the first column against the subsequent 10 columns of `tenHHexample.V.dat` it looks very similar to the plot we obtained at the end of *Tutorial 1*.

This is because none of the neurons are spiking so there are no spikes to propagate around the ring.

26.3 Providing initial stimuli

We can use a *NeuronModels::SpikeSource* to inject an initial spike into the first neuron in the ring during the first timestep to start spikes propagating. Firstly we need to define another sparse connectivity initialisation snippet at the

top of `tenHHRingModel.cc` which simply creates a single synapse on the first row of the synaptic matrix:

```
class FirstToFirst : public InitSparseConnectivitySnippet::Base

public:
    DECLARE_SNIPPET(FirstToFirst, 0);
    SET_ROW_BUILD_CODE(
        "if($(id_pre) == 0) "
        "   $(addSynapse, $(id_pre));"
        ""
        "$(endRow);");
    SET_MAX_ROW_LENGTH(1);
;
IMPLEMENT_SNIPPET(FirstToFirst);
```

We then need to add it to the network by adding the following to the end of the `modelDefinition(...)` function:

```
model.addNeuronPopulation<NeuronModels::SpikeSource>("Stim", 1, , );
model.addSynapsePopulation<WeightUpdateModels::StaticPulse,␣
↪PostsynapticModels::ExpCond>(
    "StimPop1", SynapseMatrixType::SPARSE_GLOBALG, NO_DELAY,
    "Stim", "Pop1",
    , s_ini,
    ps_p, ,
    initConnectivity<FirstToFirst>());
```

and finally inject a spike in the first timestep (in the same way that the `t` variable is provided by GeNN to keep track of the current simulation time in milliseconds, `iT` is provided to keep track of it in timesteps):

```
if(iT == 0)
    spikeCount_Stim = 1;
    spike_Stim[0] = 0;
    pushStimCurrentSpikesToDevice();
```

`spike_Stim[n]` is used to specify the indices of the neurons in population `Stim` spikes which should emit spikes where $n \in [0, \text{spikeCount\_Stim})$.

At this point our user code `tenHHRingModel.cc` should look like this

```
// Model definintion file tenHHRing.cc
#include "modelSpec.h"

class Ring : public InitSparseConnectivitySnippet::Base

public:
    DECLARE_SNIPPET(Ring, 0);
    SET_ROW_BUILD_CODE(
        "$(addSynapse, ($(id_pre) + 1) % $(num_post));"
        "$(endRow);");
    SET_MAX_ROW_LENGTH(1);
;
IMPLEMENT_SNIPPET(Ring);

class FirstToFirst : public InitSparseConnectivitySnippet::Base

public:
    DECLARE_SNIPPET(FirstToFirst, 0);
```

```
    SET_ROW_BUILD_CODE(
        "if($(id_pre) == 0) "
        "   $(addSynapse, $(id_pre));"
        ""
        "$(endRow);");
    SET_MAX_ROW_LENGTH(1);
;
IMPLEMENT_SNIPPET(FirstToFirst);

void modelDefinition(ModelSpec &model)

    // definition of tenHHRing
    model.setDT(0.1);
    model.setName("tenHHRing");

    NeuronModels::TraubMiles::ParamValues p(
        7.15,       // 0 - gNa: Na conductance in muS
        50.0,       // 1 - ENa: Na equi potential in mV
        1.43,       // 2 - gK: K conductance in muS
        -95.0,      // 3 - EK: K equi potential in mV
        0.02672,    // 4 - gl: leak conductance in muS
        -63.563,    // 5 - El: leak equi potential in mV
        0.143);     // 6 - Cmem: membr. capacity density in nF

    NeuronModels::TraubMiles::VarValues ini(
        -60.0,          // 0 - membrane potential V
        0.0529324,      // 1 - prob. for Na channel activation m
        0.3176767,      // 2 - prob. for not Na channel blocking h
        0.5961207);     // 3 - prob. for K channel activation n

    model.addNeuronPopulation<NeuronModels::TraubMiles>("Pop1", 10, p, ini);
    model.addNeuronPopulation<NeuronModels::SpikeSource>("Stim", 1, , );

    WeightUpdateModels::StaticPulse::VarValues s_ini(
         -0.2); // 0 - g: the synaptic conductance value

    PostsynapticModels::ExpCond::ParamValues ps_p(
        1.0,    // 0 - tau_S: decay time constant for S [ms]
        -80.0); // 1 - Erev: Reversal potential

    model.addSynapsePopulation<WeightUpdateModels::StaticPulse,␣
↪PostsynapticModels::ExpCond>(
        "Pop1self", SynapseMatrixType::SPARSE_GLOBALG, 100,
        "Pop1", "Pop1",
        , s_ini,
        ps_p, ,
        initConnectivity<Ring>());

    model.addSynapsePopulation<WeightUpdateModels::StaticPulse,␣
↪PostsynapticModels::ExpCond>(
        "StimPop1", SynapseMatrixType::SPARSE_GLOBALG, NO_DELAY,
        "Stim", "Pop1",
        , s_ini,
        ps_p, ,
```

```
        initConnectivity<FirstToFirst>());
```

and `tenHHRingSimulation.cc` ‘ should look like this:

```
// Standard C++ includes
#include <fstream>

// tenHHRing simulation code
#include "tenHHRing_CODE/definitions.h"

int main()

    allocateMem();
    initialize();
    initializeSparse();

    std::ofstream os("tenHHRing_output.V.dat");
    while(t < 200.0f)
        if(iT == 0)
            glbSpkStim[0] = 0;
            glbSpkCntStim[0] = 1;
            pushStimCurrentSpikesToDevice();


        stepTimeU();
        pullVPop1FromDevice();

        os << t << " ";
        for (int j= 0; j < 10; j++)
            os << VPop1[j] << " ";

        os << std::endl;

    os.close();
    return 0;
```

Finally if we build, make and run this model; and plot the first 200 ms of the ten neurons’ membrane voltages - they now looks like this:

Previous | *Top* | *Next*

CHAPTER 27

User Manual

vspace{0cm}mbox{}vspace{0cm}

27.1 Contents

27.2 Introduction

GeNN is a software library for facilitating the simulation of neuronal network models on NVIDIA CUDA enabled GPU hardware. It was designed with computational neuroscience models in mind rather than artificial neural networks. The main philosophy of GeNN is two-fold:

1. GeNN relies heavily on code generation to make it very flexible and to allow adjusting simulation code to the model of interest and the GPU hardware that is detected at compile time.
2. GeNN is lightweight in that it provides code for running models of neuronal networks on GPU hardware but it leaves it to the user to write a final simulation engine. It so allows maximal flexibility to the user who can use any of the provided code but can fully choose, inspect, extend or otherwise modify the generated code. They

can also introduce their own optimisations and in particular control the data flow from and to the GPU in any desired granularity.

This manual gives an overview of how to use GeNN for a novice user and tries to lead the user to more expert use later on. With that we jump right in.

Previous | *Top* | *Next*

27.2.1 Current source models

There is a number of predefined models which can be used with the *ModelSpec::addCurrentSource* function:

- *CurrentSourceModels::DC*
- *CurrentSourceModels::GaussianNoise*

Defining your own current source model

In order to define a new current source type for use in a GeNN application, it is necessary to define a new class derived from *CurrentSourceModels::Base*. For convenience the methods this class should implement can be implemented using macros:

- *DECLARE_MODEL(TYPE, NUM_PARAMS, NUM_VARS)*, *SET_DERIVED_PARAMS()*, *SET_PARAM_NAMES()*, *SET_VARS()* perform the same roles as they do in the neuron models discussed in *Defining your own neuron type*.
- SET_INJECTION_CODE(INJECTION_CODE) : where INJECTION_CODE contains the code for injecting current into the neuron every simulation timestep. The $(injectCurrent,) function is used to inject current.

For example, using these macros, we can define a uniformly distributed noisy current source:

```
class UniformNoise : public CurrentSourceModels::Base

public:
    DECLARE_MODEL(UniformNoise, 1, 0);

    SET_SIM_CODE("$(injectCurrent, $(gennrand_uniform) * $(magnitude));");

    SET_PARAM_NAMES("magnitude");
;
```

Previous | *Top* | *Next*

27.2.2 Defining a network model

A network model is defined by the user by providing the function

```
void modelDefinition(ModelSpec &model)
```

in a separate file, such as `MyModel.cc`. In this function, the following tasks must be completed:

1. The name of the model must be defined:

   ```
   model.setName("MyModel");
   ```

2. Neuron populations (at least one) must be added (see *Defining neuron populations*). The user may add as many neuron populations as they wish. If resources run out, there will not be a warning but GeNN will fail. However, before this breaking point is reached, GeNN will make all necessary efforts in terms of block size optimisation to accommodate the defined models. All populations must have a unique name.

3. Synapse populations (zero or more) can be added (see *Defining synapse populations*). Again, the number of synaptic connection populations is unlimited other than by resources.

Defining neuron populations

Neuron populations are added using the function

```
model.addNeuronPopulation<NeuronModel>(name, num, paramValues,␣
↪varInitialisers);
```

where the arguments are:

- `NeuronModel` : Template argument specifying the type of neuron model These should be derived off *NeuronModels::Base* and can either be one of the standard models or user-defined (see *Neuron models*).
- `const string &name` : Unique name of the neuron population
- `unsigned int size` : number of neurons in the population
- `NeuronModel::ParamValues paramValues` : Parameters of this neuron type
- `NeuronModel::VarValues varInitialisers` : Initial values or initialisation snippets for variables of this neuron type

The user may add as many neuron populations as the model necessitates. They must all have unique names. The possible values for the arguments, predefined models and their parameters and initial values are detailed *Neuron models* below.

Defining synapse populations

Synapse populations are added with the function

```
model.addSynapsePopulation<WeightUpdateModel, PostsynapticModel>(name, mType,␣
↪delay, preName, postName,

↪weightParamValues, weightVarValues, weightPreVarInitialisers,␣
↪weightPostVarInitialisers,

↪postsynapticParamValues, postsynapticVarValues, connectivityInitialiser);
```

where the arguments are

- `WeightUpdateModel` : Template parameter specifying the type of weight update model. These should be derived off *WeightUpdateModels::Base* and can either be one of the standard models or user-defined (see *Weight update models*).
- `PostsynapticModel` : Template parameter specifying the type of postsynaptic integration model. These should be derived off *PostsynapticModels::Base* and can either be one of the standard models or user-defined (see *Postsynaptic integration methods*).
- `const string &name` : The name of the synapse population
- `unsigned int mType` : How the synaptic matrix is stored. See *Synaptic matrix types* for available options.
- `unsigned int delay` : Homogeneous (axonal) delay for synapse population (in terms of the simulation time step `DT`).
- `const string preName` : Name of the (existing!) pre-synaptic neuron population.
- `const string postName` : Name of the (existing!) post-synaptic neuron population.

- `WeightUpdateModel::ParamValues weightParamValues` : The parameter values (common to all synapses of the population) for the weight update model.
- `WeightUpdateModel::VarValues weightVarInitialisers` : Initial values or initialisation snippets for the weight update model's state variables
- `WeightUpdateModel::PreVarValues weightPreVarInitialisers` : Initial values or initialisation snippets for the weight update model's presynaptic state variables
- `WeightUpdateModel::PostVarValues weightPostVarInitialisers` : Initial values or initialisation snippets for the weight update model's postsynaptic state variables
- `PostsynapticModel::ParamValues postsynapticParamValues` : The parameter values (common to all postsynaptic neurons) for the postsynaptic model.
- `PostsynapticModel::VarValues postsynapticVarInitialisers` : Initial values or initialisation snippets for variables for the postsynaptic model's state variables
- `InitSparseConnectivitySnippet::Init connectivityInitialiser` : Optional argument, specifying the initialisation snippet for synapse population's sparse connectivity (see *Sparse connectivity initialisation*).

The *ModelSpec::addSynapsePopulation()* function returns a pointer to the newly created *SynapseGroup* object which can be further configured, namely with:

- *SynapseGroup::setMaxConnections()* and *SynapseGroup::setMaxSourceConnections()* to configure the maximum number of rows and columns respectively allowed in the synaptic matrix - this can improve performance and reduce memory usage when using SynapseMatrixConnectivity::SPARSE connectivity (see *Synaptic matrix types*). When using a sparse connectivity initialisation snippet, these values are set automatically.
- *SynapseGroup::setMaxDendriticDelayTimesteps()* sets the maximum dendritic delay (in terms of the simulation time step `DT`) allowed for synapses in this population. No values larger than this should be passed to the delay parameter of the `addToDenDelay` function in user code (see *Defining a new weight update model*).
- *SynapseGroup::setSpanType()* sets how incoming spike processing is parallelised for this synapse group. The default SynapseGroup::SpanType::POSTSYNAPTIC is nearly always the best option, but SynapseGroup::SpanType::PRESYNAPTIC may perform better when there are large numbers of spikes every timestep or very few postsynaptic neurons.

If the synapse matrix uses one of the "GLOBALG" types then the global value of the synapse parameters are taken from the initial value provided in `weightVarInitialisers` therefore these must be constant rather than sampled from a distribution etc.

Previous | *Top* | *Next*

27.2.3 Neuron models

There is a number of predefined models which can be used with the ModelSpec::addNeuronGroup function:

- *NeuronModels::RulkovMap*
- *NeuronModels::Izhikevich*
- *NeuronModels::IzhikevichVariable*
- *NeuronModels::LIF*
- *NeuronModels::SpikeSource*
- *NeuronModels::PoissonNew*
- *NeuronModels::TraubMiles*

- *NeuronModels::TraubMilesFast*
- *NeuronModels::TraubMilesAlt*
- *NeuronModels::TraubMilesNStep*

Defining your own neuron type

In order to define a new neuron type for use in a GeNN application, it is necessary to define a new class derived from *NeuronModels::Base*. For convenience the methods this class should implement can be implemented using macros:

- *DECLARE_MODEL(TYPE, NUM_PARAMS, NUM_VARS)* : declared the boilerplate code required for the model e.g. the correct specialisations of NewModels::ValueBase used to wrap the neuron model parameters and values.
- SET_SIM_CODE(SIM_CODE) : where SIM_CODE contains the code for executing the integration of the model for one time stepWithin this code string, variables need to be referred to by $(NAME), where NAME is the name of the variable as defined in the vector varNames. The code may refer to the predefined primitives `DT` for the time step size and ‘‘‘ for the total incoming synaptic current. It can also refer to a unique ID (within the population) using .
- SET_THRESHOLD_CONDITION_CODE(THRESHOLD_CONDITION_CODE) defines the condition for true spike detection.
- *SET_PARAM_NAMES()* defines the names of the model parameters. If defined as `NAME` here, they can then be referenced as $(NAME) in the code string. The length of this list should match the NUM_PARAM specified in DECLARE_MODEL. Parameters are assumed to be always of type double.
- *SET_VARS()* defines the names and type strings (e.g. “float”, “double”, etc) of the neuron state variables. The type string “scalar” can be used for variables which should be implemented using the precision set globally for the model with *ModelSpec::setPrecision*. The variables defined here as `NAME` can then be used in the syntax $(NAME) in the code string.
- SET_NEEDS_AUTO_REFRACTORY() defines whether the neuron should include an automatic refractory period to prevent it emitting spikes in successive timesteps.

For example, using these macros, we can define a leaky integrator $\tau \frac{dV}{dt} = -V + I_{\text{syn}}$ solved using Euler's method:

```
class LeakyIntegrator : public NeuronModels::Base

public:
    DECLARE_MODEL(LeakyIntegrator, 1, 1);

    SET_SIM_CODE("$(V)+= (-$(V)+$(Isyn))*(DT/$(tau));");

    SET_THRESHOLD_CONDITION_CODE("$(V) >= 1.0");

    SET_PARAM_NAMES("tau");

    SET_VARS("V", "scalar");
;
```

Additionally “dependent parameters” can be defined. Dependent parameters are a mechanism for enhanced efficiency when running neuron models. If parameters with model-side meaning, such as time constants or conductances always appear in a certain combination in the model, then it is more efficient to pre-compute this combination and define it as a dependent parameter.

For example, because the equation defining the previous leaky integrator example has an algebraic solution, it can be more accurately solved as follows - using a derived parameter to calculate $\exp\left(\frac{-t}{\tau}\right)$:

```
class LeakyIntegrator2 : public NeuronModels::Base

public:
    DECLARE_MODEL(LeakyIntegrator2, 1, 1);

    SET_SIM_CODE("$(V) = $(Isyn) - $(ExpTC)*($(Isyn) - $(V));");

    SET_THRESHOLD_CONDITION_CODE("$(V) >= 1.0");

    SET_PARAM_NAMES("tau");

    SET_VARS("V", "scalar");

    SET_DERIVED_PARAMS(
        "ExpTC", [](const vector<double> &pars, double dt) return std::exp(-
↪dt / pars[0]); );
;
```

GeNN provides several additional features that might be useful when defining more complex neuron models.

Support code

Support code enables a code block to be defined that contains supporting code that will be utilized in multiple pieces of user code. Typically, these are functions that are needed in the sim code or threshold condition code. If possible, these should be defined as __host__ __device__ functions so that both GPU and CPU versions of GeNN code have an appropriate support code function available. The support code is protected with a namespace so that it is exclusively available for the neuron population whose neurons define it. Support code is added to a model using the SET_SUPPORT_CODE() macro, for example:

```
SET_SUPPORT_CODE("__device__ __host__ scalar mysin(float x) return sin(x); ");
```

Extra global parameters

Extra global parameters are parameters common to all neurons in the population. However, unlike the standard neuron parameters, they can be varied at runtime meaning they could, for example, be used to provide a global reward signal. These parameters are defined by using the SET_EXTRA_GLOBAL_PARAMS() macro to specify a list of variable names and type strings (like the *SET_VARS()* macro). For example:

```
SET_EXTRA_GLOBAL_PARAMS("R", "float");
```

These variables are available to all neurons in the population. They can also be used in synaptic code snippets; in this case it need to be addressed with a `_pre` or `_post` postfix.

For example, if the model with the "R" parameter was used for the pre-synaptic neuron population, the weight update model of a synapse population could have simulation code like:

```
SET_SIM_CODE("$(x)= $(x)+$(R_pre);");
```

where we have assumed that the weight update model has a variable `x` and our synapse type will only be used in conjunction with pre-synaptic neuron populations that do have the extra global parameter `R`. If the pre-synaptic population does not have the required variable/parameter, GeNN will fail when compiling the kernels.

Additional input variables

Normally, neuron models receive the linear sum of the inputs coming from all of their synaptic inputs through the $(inSyn) variable. However neuron models can define additional input variables - allowing input from different synaptic inputs to be combined non-linearly. For example, if we wanted our leaky integrator to operate on the the product of two input currents, it could be defined as follows:

```
SET_ADDITIONAL_INPUT_VARS("Isyn2", "scalar", 1.0);
SET_SIM_CODE("const scalar input = $(Isyn) * $(Isyn2);"
             "$(V) = input - $(ExpTC)*(input - $(V));");
```

Where the SET_ADDITIONAL_INPUT_VARS() macro defines the name, type and its initial value before postsynaptic inputs are applyed (see section *Postsynaptic integration methods* for more details).

Random number generation

Many neuron models have probabilistic terms, for example a source of noise or a probabilistic spiking mechanism. In GeNN this can be implemented by using the following functions in blocks of model code:

- `$(gennrand_uniform)` returns a number drawn uniformly from the interval $[0.0, 1.0]$
- `$(gennrand_normal)` returns a number drawn from a normal distribution with a mean of 0 and a standard deviation of 1.
- `$(gennrand_exponential)` returns a number drawn from an exponential distribution with $\lambda = 1$.
- `$(gennrand_log_normal, MEAN, STDDEV)` returns a number drawn from a log-normal distribution with the specified mean and standard deviation.
- `$(gennrand_gamma, ALPHA)` returns a number drawn from a gamma distribution with the specified shape.

Once defined in this way, new neuron models classes, can be used in network descriptions by referring to their type e.g.

```
networkModel.addNeuronPopulation<LeakyIntegrator>("Neurons", 1,
                                                   LeakyIntegrator::ParamValues(20.
↪0), // tau
                                                   LeakyIntegrator::VarValues(0.
↪0)); // V
```

Previous | *Top* | *Next*

27.2.4 Postsynaptic integration methods

There are currently 3 built-in postsynaptic integration methods:

- *PostsynapticModels::ExpCurr*
- *PostsynapticModels::ExpCond*
- *PostsynapticModels::DeltaCurr*

Defining a new postsynaptic model

The postsynaptic model defines how synaptic activation translates into an input current (or other input term for models that are not current based). It also can contain equations defining dynamics that are applied to the (summed) synaptic activation, e.g. an exponential decay over time.

In the same manner as to both the neuron and weight update models discussed in *Defining your own neuron type* and *Defining a new weight update model*, postsynamic model definitions are encapsulated in a class derived from *PostsynapticModels::Base*. Again, the methods that a postsynaptic model should implement can be implemented using the following macros:

- *DECLARE_MODEL(TYPE, NUM_PARAMS, NUM_VARS)*, *SET_DERIVED_PARAMS()*, *SET_PARAM_NAMES()*, *SET_VARS()* perform the same roles as they do in the neuron models discussed in *Defining your own neuron type*.
- SET_DECAY_CODE(DECAY_CODE) defines the code which provides the continuous time dynamics for the summed presynaptic inputs to the postsynaptic neuron. This usually consists of some kind of decay function.
- SET_APPLY_INPUT_CODE(APPLY_INPUT_CODE) defines the code specifying the conversion from synaptic inputs to a postsynaptic neuron input current. e.g. for a conductance model:

  ```
  SET_APPLY_INPUT_CODE("$(Isyn) += $(inSyn) * ($(E) - $(V))");
  ```

  where $(E) is a postsynaptic model parameter specifying reversal potential and $(V) is the variable containing the postsynaptic neuron's membrane potential. As discussed in *Built-in Variables in GeNN*, $(Isyn) is the built in variable used to sum neuron input. However additional input variables can be added to a neuron model using the SET_ADDITIONAL_INPUT_VARS() macro (see *Defining your own neuron type* for more details).

Previous | *Top* | *Next*

27.2.5 Sparse connectivity initialisation

Synaptic connectivity implemented using SynapseMatrixConnectivity::SPARSE and SynapseMatrixConnectivity::BITMASK can be automatically initialised.

This can be done using one of a number of predefined *sparse connectivity initialisation snippets* :

- *InitSparseConnectivitySnippet::OneToOne*
- *InitSparseConnectivitySnippet::FixedProbability*
- *InitSparseConnectivitySnippet::FixedProbabilityNoAutapse*

For example, to initialise synaptic connectivity with a 10% connection probability (allowing connections between neurons with the same id):

```
InitSparseConnectivitySnippet::FixedProbability::ParamValues fixedProb(0.1);

model.addSynapsePopulation<...>(
        ...
        initConnectivity<InitSparseConnectivitySnippet::FixedProbability>(fixedProb));
```

Defining a new sparse connectivity snippet

Similarly to variable initialisation snippets, sparse connectivity initialisation snippets can be created by simply defining a class in the model description.

For example, the following sparse connectivity initialisation snippet could be used to initialise a 'ring' of connectivity where each neuron is connected to a number of subsequent neurons specified using the `numNeighbours` parameter:

```
class Ring : public InitSparseConnectivitySnippet::Base

public:
    DECLARE_SNIPPET(Ring, 1);
```

```
    SET_ROW_BUILD_STATE_VARS("offset", "unsigned int", 1);
    SET_ROW_BUILD_CODE(
        "const unsigned int target = ($(id_pre) + offset) % $(num_post);"
        "$(addSynapse, target);"
        "offset++;"
        "if(offset > (unsigned int)$(numNeighbours)) "
        "   $(endRow);"
        "");

    SET_PARAM_NAMES("numNeighbours");
    SET_CALC_MAX_ROW_LENGTH_FUNC(
        [](unsigned int numPre, unsigned int numPost, const␣
↪std::vector<double> &pars)

            return (unsigned int)pars[0];
        );
    SET_CALC_MAX_COL_LENGTH_FUNC(
        [](unsigned int numPre, unsigned int numPost, const␣
↪std::vector<double> &pars)

            return (unsigned int)pars[0];
        );
;
IMPLEMENT_SNIPPET(Ring);
```

Each *row* of sparse connectivity is initialised independantly by running the snippet of code specified using the *SET_ROW_BUILD_CODE()* macro within a loop. The $(num_post) variable can be used to access the number of neurons in the postsynaptic population and the $(id_pre) variable can be used to access the index of the presynaptic neuron associated with the row being generated. The *SET_ROW_BUILD_STATE_VARS()* macro can be used to initialise state variables outside of the loop - in this case `offset` which is used to count the number of synapses created in each row. Synapses are added to the row using the $(addSynapse, target) function and iteration is stopped using the $(endRow) function. To avoid having to manually call *SynapseGroup::setMaxConnections* and *SynapseGroup::setMaxSourceConnections*, sparse connectivity snippets can also provide code to calculate the maximum row and column lengths this connectivity will result in using the SET_CALC_MAX_ROW_LENGTH_FUNC() and SET_CALC_MAX_COL_LENGTH_FUNC() macros. Alternatively, if the maximum row or column length is constant, the *SET_MAX_ROW_LENGTH()* and *SET_MAX_COL_LENGTH()* shorthand macros can be used.

Sparse connectivity locations

Once you have defined **how** sparse connectivity is going to be initialised, similarly to variables, you can control **where** it is allocated. This is controlled using the same `VarLocations` options described in section *Variable locations* and can either be set using the model default specifiued with `ModelSpec::setDefaultSparseConnectivityLocation` or on a per-synapse group basis using `SynapseGroup::setSparseConnectivityLocation`.

Previous | *Top* | *Next*

27.2.6 Synaptic matrix types

Synaptic matrix types are made up of two components: SynapseMatrixConnectivity and SynapseMatrixWeight. SynapseMatrixConnectivity defines what data structure is used to store the synaptic matrix:

- SynapseMatrixConnectivity::DENSE stores synaptic matrices as a dense matrix. Large dense matrices require a large amount of memory and if they contain a lot of zeros it may be inefficient.
- SynapseMatrixConnectivity::SPARSE stores synaptic matrices in a(padded) 'ragged array' format. In general, this is less efficient to traverse using a GPU than the dense matrix format but does result in significant memory savings for large matrices. Ragged matrix connectivity is stored using several variables whose names, like state variables, have the name of the synapse population appended to them:

    1. `const unsigned int maxRowLength` : a constant set via the `SynapseGroup::setMaxConnections` method which specifies the maximum number of connections in any given row (this is the width the structure is padded to).
    2. `unsigned int *rowLength` (sized to number of presynaptic neurons): actual length of the row of connections associated with each presynaptic neuron
    3. `unsigned int *ind` (sized to `maxRowLength * number of presynaptic neurons`): Indices of corresponding postsynaptic neurons concatenated for each presynaptic neuron. For example, consider a network of two presynaptic neurons connected to three postsynaptic neurons: 0th presynaptic neuron connected to 1st and 2nd postsynaptic neurons, the 1st presynaptic neuron connected only to the 0th neuron. The struct RaggedProjection should have these members, with indexing from 0 (where X represents a padding value):

        ```
        maxRowLength = 2
        ind = [1 2 0 X]
        rowLength = [2 1]
        ```

        Weight update model variables associated with the sparsely connected synaptic population will be kept in an array using the same indexing as ind. For example, a variable caled `g` will be kept in an array such as: `g`= [g_Pre0-Post1 g_pre0-post2 g_pre1-post0 X]
- SynapseMatrixConnectivity::BITMASK is an alternative sparse matrix implementation where which synapses within the matrix are present is specified as a binary array (see *Insect olfaction model*). This structure is somewhat less efficient than the *`SynapseMatrixConnectivity::SPARSE`* and `SynapseMatrixConnectivity::RAGGED` formats and doesn't allow individual weights per synapse. However it does require the smallest amount of GPU memory for large networks.

Furthermore the SynapseMatrixWeight defines how

- SynapseMatrixWeight::INDIVIDUAL allows each individual synapse to have unique weight update model variables. Their values must be initialised at runtime and, if running on the GPU, copied across from the user side code, using the `pushXXXXXStateToDevice` function, where XXXX is the name of the synapse population.
- SynapseMatrixWeight::INDIVIDUAL_PSM allows each postsynapic neuron to have unique post synaptic model variables. Their values must be initialised at runtime and, if running on the GPU, copied across from the user side code, using the `pushXXXXXStateToDevice` function, where XXXX is the name of the synapse population.
- SynapseMatrixWeight::GLOBAL saves memory by only maintaining one copy of the weight update model variables. This is automatically initialized to the initial value passed to *ModelSpec::addSynapsePopulation*.

Only certain combinations of SynapseMatrixConnectivity and SynapseMatrixWeight are sensible therefore, to reduce confusion, the SynapseMatrixType enumeration defines the following options which can be passed to *ModelSpec::addSynapsePopulation* :

- SynapseMatrixType::SPARSE_GLOBALG
- SynapseMatrixType::SPARSE_GLOBALG_INDIVIDUAL_PSM
- SynapseMatrixType::SPARSE_INDIVIDUALG
- SynapseMatrixType::DENSE_GLOBALG

- SynapseMatrixType::DENSE_GLOBALG_INDIVIDUAL_PSM
- SynapseMatrixType::DENSE_INDIVIDUALG
- SynapseMatrixType::BITMASK_GLOBALG
- SynapseMatrixType::BITMASK_GLOBALG_INDIVIDUAL_PSM

Previous | *Top* | *Next*

27.2.7 Variable initialisation

Neuron, weight update and postsynaptic models all have state variables which GeNN can automatically initialise.

Previously we have shown variables being initialised to constant values such as:

```
NeuronModels::TraubMiles::VarValues ini(
    0.0529324,     // 1 - prob. for Na channel activation m
    ...
);
```

state variables can also be left *uninitialised* leaving it up to the user code to initialise them between the calls to `initialize()` and `initializeSparse()`:

```
NeuronModels::TraubMiles::VarValues ini(
    uninitialisedVar(),     // 1 - prob. for Na channel activation m
    ...
);
```

or initialised using one of a number of predefined *variable initialisation snippets* :

- *InitVarSnippet::Uniform*
- *InitVarSnippet::Normal*
- *InitVarSnippet::Exponential*
- *InitVarSnippet::Gamma*

For example, to initialise a parameter using values drawn from the normal distribution:

```
InitVarSnippet::Normal::ParamValues params(
    0.05,   // 0 - mean
    0.01);  // 1 - standard deviation

NeuronModels::TraubMiles::VarValues ini(
    initVar<InitVarSnippet::Normal>(params),     // 1 - prob. for Na channel␣
↪activation m
    ...
);
```

Defining a new variable initialisation snippet

Similarly to neuron, weight update and postsynaptic models, new variable initialisation snippets can be created by simply defining a class in the model description. For example, when initialising excitatory (positive) synaptic weights with a normal distribution they should be clipped at 0 so the long tail of the normal distribution doesn't result in negative weights. This could be implemented using the following variable initialisation snippet which redraws until samples are within the desired bounds:

```
class NormalPositive : public InitVarSnippet::Base

public:
    DECLARE_SNIPPET(NormalPositive, 2);

    SET_CODE(
        "scalar normal;"
        "do"
        ""
        "   normal = $(mean) + ($(gennrand_normal) * $(sd));"
        " while (normal < 0.0);"
        "$(value) = normal;");

    SET_PARAM_NAMES("mean", "sd");
;
IMPLEMENT_SNIPPET(NormalPositive);
```

Within the snippet of code specified using the *SET_CODE()* macro, when initialisising neuron and postaynaptic model state variables , the $(id) variable can be used to access the id of the neuron being initialised. Similarly, when initialising weight update model state variables, the $(id_pre) and $(id_post) variables can used to access the ids of the pre and postsynaptic neurons connected by the synapse being initialised.

Variable locations

Once you have defined **how** your variables are going to be initialised you need to configure **where** they will be allocated. By default memory is allocated for variables on both the GPU and the host. However, the following alternative 'variable locations' are available:

- VarLocation::DEVICE - Variables are only allocated on the GPU, saving memory but meaning that they can't easily be copied to the host - best for internal state variables.
- VarLocation::HOST_DEVICE - Variables are allocated on both the GPU and the host - the default.
- VarLocation::HOST_DEVICE_ZERO_COPY - Variables are allocated as 'zero-copy' memory accessible to the host and GPU - useful on devices such as Jetson TX1 where physical memory is shared between the GPU and CPU.

'Zero copy' memory is only supported on newer embedded systems such as the Jetson TX1 where there is no physical seperation between GPU and host memory and thus the same block of memory can be shared between them.

These modes can be set as a model default using `ModelSpec::setDefaultVarLocation` or on a per-variable basis using one of the following functions:

- *NeuronGroup::setSpikeLocation*
- *NeuronGroup::setSpikeEventLocation*
- *NeuronGroup::setSpikeTimeLocation*
- *NeuronGroup::setVarLocation*
- *SynapseGroup::setWUVarLocation*
- *SynapseGroup::setWUPreVarLocation*
- *SynapseGroup::setWUPostVarLocation*
- *SynapseGroup::setPSVarLocation*
- *SynapseGroup::setInSynVarLocation*

Previous | *Top* | *Next*

27.2.8 Weight update models

Currently 4 predefined weight update models are available:

- *WeightUpdateModels::StaticPulse*
- *WeightUpdateModels::StaticPulseDendriticDelay*
- *WeightUpdateModels::StaticGraded*
- *WeightUpdateModels::PiecewiseSTDP*

For more details about these built-in synapse models, see Nowotny2010.

Defining a new weight update model

Like the neuron models discussed in *Defining your own neuron type*, new weight update models are created by defining a class. Weight update models should all be derived from WeightUpdateModel::Base and, for convenience, the methods a new weight update model should implement can be implemented using macros:

- *SET_DERIVED_PARAMS()*, *SET_PARAM_NAMES()*, *SET_VARS()* and SET_EXTRA_GLOBAL_PARAMS() perform the same roles as they do in the neuron models discussed in *Defining your own neuron type*.
- DECLARE_WEIGHT_UPDATE_MODEL(TYPE, NUM_PARAMS, NUM_VARS, NUM_PRE_VARS, NUM_POST_VARS) is an extended version of *DECLARE_MODEL()* which declares the boilerplate code required for a weight update model with pre and postsynaptic as well as per-synapse state variables.
- SET_PRE_VARS() and SET_POST_VARS() define state variables associated with pre or postsynaptic neurons rather than synapses. These are typically used to efficiently implement *trace* variables for use in STDP learning rules Morrison2008. Like other state variables, variables defined here as `NAME` can be accessed in weight update model code strings using the $(NAME) syntax.
- SET_SIM_CODE(SIM_CODE) : defines the simulation code that is used when a true spike is detected. The update is performed only in timesteps after a neuron in the presynaptic population has fulfilled its threshold detection condition. Typically, spikes lead to update of synaptic variables that then lead to the activation of input into the post-synaptic neuron. Most of the time these inputs add linearly at the post-synaptic neuron. This is assumed in GeNN and the term to be added to the activation of the post-synaptic neuron should be applied using the the $(addToInSyn, weight) function. For example

  ```
  SET_SIM_CODE(
      "$(addToInSyn, $(inc));"
  ```

  where "inc" is the increment of the synaptic input to a post-synaptic neuron for each pre-synaptic spike. The simulation code also typically contains updates to the internal synapse variables that may have contributed to . For an example, see *WeightUpdateModels::StaticPulse* for a simple synapse update model and *WeightUpdateModels::PiecewiseSTDP* for a more complicated model that uses STDP. To apply input to the post-synaptic neuron with a dendritic (i.e. between the synapse and the postsynaptic neuron) delay you can instead use the $(addToInSynDelay, weight, delay) function. For example

  ```
  SET_SIM_CODE(
      "$(addToInSynDelay, $(inc), $(delay));");
  ```

  where, once again, `inc` is the magnitude of the input step to apply and `delay` is the length of the dendritic delay in timesteps. By implementing `delay` as a weight update model variable, heterogeneous synaptic delays can be implemented. For an example, see *WeightUpdateModels::StaticPulseDendriticDelay* for a simple synapse update model with heterogeneous dendritic delays. When

using dendritic delays, the **maximum** dendritic delay for a synapse populations must be specified using the `SynapseGroup::setMaxDendriticDelayTimesteps()` function.

- SET_EVENT_THRESHOLD_CONDITION_CODE(EVENT_THRESHOLD_CONDITION_CODE) defines a condition for a synaptic event. This typically involves the pre-synaptic variables, e.g. the membrane potential:

  ```
  SET_EVENT_THRESHOLD_CONDITION_CODE("$(V_pre) > -0.02");
  ```

  Whenever this expression evaluates to true, the event code set using the SET_EVENT_CODE() macro is executed. For an example, see *WeightUpdateModels::StaticGraded*.

- SET_EVENT_CODE(EVENT_CODE) defines the code that is used when the event threshold condition is met (as set using the SET_EVENT_THRESHOLD_CONDITION_CODE() macro).

- SET_LEARN_POST_CODE(LEARN_POST_CODE) defines the code which is used in the learnSynapsesPost kernel/function, which performs updates to synapses that are triggered by post-synaptic spikes. This is typically used in STDP-like models e.g. *WeightUpdateModels::PiecewiseSTDP*.

- SET_SYNAPSE_DYNAMICS_CODE(SYNAPSE_DYNAMICS_CODE) defines code that is run for each synapse, each timestep i.e. unlike the others it is not event driven. This can be used where synapses have internal variables and dynamics that are described in continuous time, e.g. by ODEs. However using this mechanism is typically computationally very costly because of the large number of synapses in a typical network. By using the $(addtoinsyn), $(updatelinsyn) and $(addToDenDelay) mechanisms discussed in the context of SET_SIM_CODE(), the synapse dynamics can also be used to implement continuous synapses for rate-based models.

- SET_PRE_SPIKE_CODE() and SET_POST_SPIKE_CODE() define code that is called whenever there is a pre or postsynaptic spike. Typically these code strings are used to update any pre or postsynaptic state variables.

- SET_NEEDS_PRE_SPIKE_TIME(PRE_SPIKE_TIME_REQUIRED) and SET_NEEDS_POST_SPIKE_TIME(POST_SPIKE_T define whether the weight update needs to know the times of the spikes emitted from the pre and postsynaptic populations. For example an STDP rule would be likely to require:

  ```
  SET_NEEDS_PRE_SPIKE_TIME(true);
  SET_NEEDS_POST_SPIKE_TIME(true);
  ```

All code snippets, aside from those defined with *SET_PRE_SPIKE_CODE()* and *SET_POST_SPIKE_CODE()*, can be used to manipulate any synapse variable and so learning rules can combine both time-drive and event-driven processes.

Previous | *Top* | *Next*

Related Pages:

Current source models
Defining a network model
Neuron models
Postsynaptic integration methods
Sparse connectivity initialisation
Synaptic matrix types
Variable initialisation
Weight update models

Related Pages:

Best practices guide

Reference and Index:

CHAPTER 28

Global Namespace

28.1 namespace CodeGenerator

28.1.1 namespace CodeGenerator::CUDA

namespace CodeGenerator::CUDA::Optimiser

```
namespace Optimiser

// global functions

BACKEND_EXPORT Backend createBackend(
    const ModelSpecInternal& model,
    const filesystem::path& outputPath,
    int localHostID,
    const Preferences& preferences
    );

 // namespace Optimiser
```

namespace CodeGenerator::CUDA::PresynapticUpdateStrategy

class CodeGenerator::CUDA::PresynapticUpdateStrategy::Base

Overview

```
#include <presynapticUpdateStrategy.h>

class Base
```

```
public:
    // methods

    virtual size_t getNumThreads(const SynapseGroupInternal& sg) const = 0;
    virtual bool isCompatible(const SynapseGroupInternal& sg) const = 0;
    virtual bool shouldAccumulateInRegister(const SynapseGroupInternal& sg,␣
↪const Backend& backend) const = 0;
    virtual bool shouldAccumulateInSharedMemory(const SynapseGroupInternal&␣
↪sg, const Backend& backend) const = 0;

    virtual void genCode(
        CodeStream& os,
        const ModelSpecInternal& model,
        const SynapseGroupInternal& sg,
        const Substitutions& popSubs,
        const Backend& backend,
        bool trueSpike,
        BackendBase::SynapseGroupHandler wumThreshHandler,
        BackendBase::SynapseGroupHandler wumSimHandler
        ) const = 0;
;

// direct descendants

class PostSpan;
class PreSpan;
```

Detailed Documentation

Methods

`virtual size_t getNumThreads(const SynapseGroupInternal& sg) const = 0`

Get the number of threads that presynaptic updates should be parallelised across.

`virtual bool isCompatible(const SynapseGroupInternal& sg) const = 0`

Is this presynaptic update strategy compatible with a given synapse group?

`virtual bool shouldAccumulateInRegister(const SynapseGroupInternal& sg, const␣↪Backend& backend) const = 0`

Are input currents emitted by this presynaptic update accumulated into a register?

`virtual bool shouldAccumulateInSharedMemory(const SynapseGroupInternal& sg,␣↪const Backend& backend) const = 0`

Are input currents emitted by this presynaptic update accumulated into a shared memory array?

```
virtual void genCode(
    CodeStream& os,
    const ModelSpecInternal& model,
    const SynapseGroupInternal& sg,
    const Substitutions& popSubs,
    const Backend& backend,
    bool trueSpike,
    BackendBase::SynapseGroupHandler wumThreshHandler,
```

```
    BackendBase::SynapseGroupHandler wumSimHandler
    ) const = 0
```

Generate presynaptic update code.

class CodeGenerator::CUDA::PresynapticUpdateStrategy::PostSpan

Overview

Postsynaptic parallelism. *More...*

```
#include <presynapticUpdateStrategy.h>

class PostSpan: public CodeGenerator::CUDA::PresynapticUpdateStrategy::Base

public:
    // methods

    virtual size_t getNumThreads(const SynapseGroupInternal& sg) const;
    virtual bool isCompatible(const SynapseGroupInternal& sg) const;
    virtual bool shouldAccumulateInRegister(const SynapseGroupInternal& sg,␣
↪const Backend& backend) const;
    virtual bool shouldAccumulateInSharedMemory(const SynapseGroupInternal&␣
↪sg, const Backend& backend) const;

    virtual void genCode(
        CodeStream& os,
        const ModelSpecInternal& model,
        const SynapseGroupInternal& sg,
        const Substitutions& popSubs,
        const Backend& backend,
        bool trueSpike,
        BackendBase::SynapseGroupHandler wumThreshHandler,
        BackendBase::SynapseGroupHandler wumSimHandler
        ) const;
;
```

Inherited Members

```
public:
    // methods

    virtual size_t getNumThreads(const SynapseGroupInternal& sg) const = 0;
    virtual bool isCompatible(const SynapseGroupInternal& sg) const = 0;
    virtual bool shouldAccumulateInRegister(const SynapseGroupInternal& sg,␣
↪const Backend& backend) const = 0;
    virtual bool shouldAccumulateInSharedMemory(const SynapseGroupInternal&␣
↪sg, const Backend& backend) const = 0;

    virtual void genCode(
        CodeStream& os,
        const ModelSpecInternal& model,
        const SynapseGroupInternal& sg,
```

```
        const Substitutions& popSubs,
        const Backend& backend,
        bool trueSpike,
        BackendBase::SynapseGroupHandler wumThreshHandler,
        BackendBase::SynapseGroupHandler wumSimHandler
        ) const = 0;
```

Detailed Documentation

Postsynaptic parallelism.

Methods

```
virtual size_t getNumThreads(const SynapseGroupInternal& sg) const
```

Get the number of threads that presynaptic updates should be parallelised across.

```
virtual bool isCompatible(const SynapseGroupInternal& sg) const
```

Is this presynaptic update strategy compatible with a given synapse group?

```
virtual bool shouldAccumulateInRegister(const SynapseGroupInternal& sg, const␣
↪Backend& backend) const
```

Are input currents emitted by this presynaptic update accumulated into a register?

```
virtual bool shouldAccumulateInSharedMemory(const SynapseGroupInternal& sg,␣
↪const Backend& backend) const
```

Are input currents emitted by this presynaptic update accumulated into a shared memory array?

```
virtual void genCode(
    CodeStream& os,
    const ModelSpecInternal& model,
    const SynapseGroupInternal& sg,
    const Substitutions& popSubs,
    const Backend& backend,
    bool trueSpike,
    BackendBase::SynapseGroupHandler wumThreshHandler,
    BackendBase::SynapseGroupHandler wumSimHandler
    ) const
```

Generate presynaptic update code.

class CodeGenerator::CUDA::PresynapticUpdateStrategy::PreSpan

Overview

Presynaptic parallelism. *More...*

```
#include <presynapticUpdateStrategy.h>

class PreSpan: public CodeGenerator::CUDA::PresynapticUpdateStrategy::Base

public:
    // methods
```

```
    virtual size_t getNumThreads(const SynapseGroupInternal& sg) const;
    virtual bool isCompatible(const SynapseGroupInternal& sg) const;
    virtual bool shouldAccumulateInRegister(const SynapseGroupInternal& sg,␣
↪const Backend& backend) const;
    virtual bool shouldAccumulateInSharedMemory(const SynapseGroupInternal&␣
↪sg, const Backend& backend) const;

    virtual void genCode(
        CodeStream& os,
        const ModelSpecInternal& model,
        const SynapseGroupInternal& sg,
        const Substitutions& popSubs,
        const Backend& backend,
        bool trueSpike,
        BackendBase::SynapseGroupHandler wumThreshHandler,
        BackendBase::SynapseGroupHandler wumSimHandler
        ) const;
;
```

Inherited Members

```
public:
    // methods

    virtual size_t getNumThreads(const SynapseGroupInternal& sg) const = 0;
    virtual bool isCompatible(const SynapseGroupInternal& sg) const = 0;
    virtual bool shouldAccumulateInRegister(const SynapseGroupInternal& sg,␣
↪const Backend& backend) const = 0;
    virtual bool shouldAccumulateInSharedMemory(const SynapseGroupInternal&␣
↪sg, const Backend& backend) const = 0;

    virtual void genCode(
        CodeStream& os,
        const ModelSpecInternal& model,
        const SynapseGroupInternal& sg,
        const Substitutions& popSubs,
        const Backend& backend,
        bool trueSpike,
        BackendBase::SynapseGroupHandler wumThreshHandler,
        BackendBase::SynapseGroupHandler wumSimHandler
        ) const = 0;
```

Detailed Documentation

Presynaptic parallelism.

Methods

```
virtual size_t getNumThreads(const SynapseGroupInternal& sg) const
```

Get the number of threads that presynaptic updates should be parallelised across.

```
virtual bool isCompatible(const SynapseGroupInternal& sg) const
```

Is this presynaptic update strategy compatible with a given synapse group?

```
virtual bool shouldAccumulateInRegister(const SynapseGroupInternal& sg, const␣
↪Backend& backend) const
```

Are input currents emitted by this presynaptic update accumulated into a register?

```
virtual bool shouldAccumulateInSharedMemory(const SynapseGroupInternal& sg,␣
↪const Backend& backend) const
```

Are input currents emitted by this presynaptic update accumulated into a shared memory array?

```
virtual void genCode(
    CodeStream& os,
    const ModelSpecInternal& model,
    const SynapseGroupInternal& sg,
    const Substitutions& popSubs,
    const Backend& backend,
    bool trueSpike,
    BackendBase::SynapseGroupHandler wumThreshHandler,
    BackendBase::SynapseGroupHandler wumSimHandler
    ) const
```

Generate presynaptic update code.

```
namespace PresynapticUpdateStrategy

// classes

class Base;
class PostSpan;
class PreSpan;

 // namespace PresynapticUpdateStrategy
```

namespace CodeGenerator::CUDA::Utils

```
namespace Utils

// global functions

size_t ceilDivide(
    size_t numerator,
    size_t denominator
    );

size_t padSize(
    size_t size,
    size_t blockSize
    );

 // namespace Utils
```

enum CodeGenerator::CUDA::BlockSizeSelect

Overview

Methods for selecting *CUDA* kernel block size. *More...*

```
#include <backend.h>

enum BlockSizeSelect

    OCCUPANCY,
    MANUAL,
;
```

Detailed Documentation

Methods for selecting *CUDA* kernel block size.

Enum Values

`OCCUPANCY`

Pick optimal blocksize for each kernel based on occupancy.

`MANUAL`

Use block sizes specified by user.

enum CodeGenerator::CUDA::DeviceSelect

Overview

Methods for selecting *CUDA* device. *More...*

```
#include <backend.h>

enum DeviceSelect

    OPTIMAL,
    MOST_MEMORY,
    MANUAL,
;
```

Detailed Documentation

Methods for selecting *CUDA* device.

Enum Values

`OPTIMAL`

Pick optimal device based on how well kernels can be simultaneously simulated and occupancy.

`MOST_MEMORY`

Pick device with most global memory.

`MANUAL`

Use device specified by user.

enum CodeGenerator::CUDA::Kernel

Kernels generated by *CUDA* backend.

```
#include <backend.h>

enum Kernel

    KernelNeuronUpdate,
    KernelPresynapticUpdate,
    KernelPostsynapticUpdate,
    KernelSynapseDynamicsUpdate,
    KernelInitialize,
    KernelInitializeSparse,
    KernelPreNeuronReset,
    KernelPreSynapseReset,
    KernelMax,
;
```

struct CodeGenerator::CUDA::Preferences

Overview

Preferences for *CUDA* backend. *More...*

```
#include <backend.h>

struct Preferences: public CodeGenerator::PreferencesBase

    // fields

    bool showPtxInfo;
    DeviceSelect deviceSelectMethod;
    unsigned int manualDeviceID;
    BlockSizeSelect blockSizeSelectMethod;
    KernelBlockSize manualBlockSizes;
    std::string userNvccFlags;

    // methods

    Preferences();
;
```

Inherited Members

```
public:
    // fields
```

```
    bool optimizeCode;
    bool debugCode;
    std::string userCxxFlagsGNU;
    std::string userNvccFlagsGNU;
    plog::Severity logLevel;
```

Detailed Documentation

Preferences for *CUDA* backend.

Fields

`bool showPtxInfo`

Should PTX assembler information be displayed for each *CUDA* kernel during compilation.

`DeviceSelect deviceSelectMethod`

How to select GPU device.

`unsigned int manualDeviceID`

If device select method is set to *DeviceSelect::MANUAL*, id of device to use.

`BlockSizeSelect blockSizeSelectMethod`

How to select *CUDA* blocksize.

`KernelBlockSize manualBlockSizes`

If block size select method is set to *BlockSizeSelect::MANUAL*, block size to use for each kernel.

`std::string userNvccFlags`

NVCC compiler options for all GPU code.

class CodeGenerator::CUDA::Backend

Overview

```
#include <backend.h>

class Backend: public CodeGenerator::BackendBase

public:
    // fields

    static const char* KernelNames[KernelMax];

    // methods

    Backend(
        const KernelBlockSize& kernelBlockSizes,
        const Preferences& preferences,
        int localHostID,
```

```
        const std::string& scalarType,
        int device
        );

    virtual void genNeuronUpdate(CodeStream& os, const ModelSpecInternal&␣
↪model, NeuronGroupSimHandler simHandler, NeuronGroupHandler␣
↪wuVarUpdateHandler) const;

    virtual void genSynapseUpdate(
        CodeStream& os,
        const ModelSpecInternal& model,
        SynapseGroupHandler wumThreshHandler,
        SynapseGroupHandler wumSimHandler,
        SynapseGroupHandler wumEventHandler,
        SynapseGroupHandler postLearnHandler,
        SynapseGroupHandler synapseDynamicsHandler
        ) const;

    virtual void genInit(
        CodeStream& os,
        const ModelSpecInternal& model,
        NeuronGroupHandler localNGHandler,
        NeuronGroupHandler remoteNGHandler,
        SynapseGroupHandler sgDenseInitHandler,
        SynapseGroupHandler sgSparseConnectHandler,
        SynapseGroupHandler sgSparseInitHandler
        ) const;

    virtual void genDefinitionsPreamble(CodeStream& os) const;
    virtual void genDefinitionsInternalPreamble(CodeStream& os) const;
    virtual void genRunnerPreamble(CodeStream& os) const;
    virtual void genAllocateMemPreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const;
    virtual void genStepTimeFinalisePreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const;

    virtual void genVariableDefinition(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    virtual void genVariableImplementation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    virtual MemAlloc genVariableAllocation(
        CodeStream& os,
        const std::string& type,
```

```
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const;

    virtual void genVariableFree(
        CodeStream& os,
        const std::string& name,
        VarLocation loc
        ) const;

    virtual void genExtraGlobalParamDefinition(
        CodeStream& definitions,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    virtual void genExtraGlobalParamImplementation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    virtual void genExtraGlobalParamAllocation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    virtual void genExtraGlobalParamPush(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    virtual void genExtraGlobalParamPull(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    virtual void genPopVariableInit(
        CodeStream& os,
        VarLocation loc,
        const Substitutions& kernelSubs,
        Handler handler
        ) const;

    virtual void genVariableInit(
```

```
        CodeStream& os,
        VarLocation loc,
        size_t count,
        const std::string& indexVarName,
        const Substitutions& kernelSubs,
        Handler handler
        ) const;

    virtual void genSynapseVariableRowInit(
        CodeStream& os,
        VarLocation loc,
        const SynapseGroupInternal& sg,
        const Substitutions& kernelSubs,
        Handler handler
        ) const;

    virtual void genVariablePush(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        bool autoInitialized,
        size_t count
        ) const;

    virtual void genVariablePull(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const;

    virtual void genCurrentTrueSpikePush(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const;

    virtual void genCurrentTrueSpikePull(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const;

    virtual void genCurrentSpikeLikeEventPush(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const;

    virtual void genCurrentSpikeLikeEventPull(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const;

    virtual MemAlloc genGlobalRNG(
```

```
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const ModelSpecInternal& model
        ) const;

    virtual MemAlloc genPopulationRNG(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const std::string& name,
        size_t count
        ) const;

    virtual void genTimer(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        CodeStream& stepTimeFinalise,
        const std::string& name,
        bool updateInStepTime
        ) const;

    virtual void genMakefilePreamble(std::ostream& os) const;
    virtual void genMakefileLinkRule(std::ostream& os) const;
    virtual void genMakefileCompileRule(std::ostream& os) const;
    virtual void genMSBuildConfigProperties(std::ostream& os) const;
    virtual void genMSBuildImportProps(std::ostream& os) const;
    virtual void genMSBuildItemDefinitions(std::ostream& os) const;

    virtual void genMSBuildCompileModule(
        const std::string& moduleName,
        std::ostream& os
        ) const;

    virtual void genMSBuildImportTarget(std::ostream& os) const;
    virtual std::string getVarPrefix() const;
    virtual bool isGlobalRNGRequired(const ModelSpecInternal& model) const;
    virtual bool isSynRemapRequired() const;
    virtual bool isPostsynapticRemapRequired() const;
    virtual size_t getDeviceMemoryBytes() const;
    const cudaDeviceProp& getChosenCUDADevice() const;
    int getChosenDeviceID() const;
    int getRuntimeVersion() const;
    std::string getNVCCFlags() const;
    std::string getFloatAtomicAdd(const std::string& ftype) const;
    size_t getKernelBlockSize(Kernel kernel) const;
    static size_t getNumPresynapticUpdateThreads(const SynapseGroupInternal&␣
```

```
↪sg);
    static size_t getNumPostsynapticUpdateThreads(const SynapseGroupInternal&␣
↪sg);
    static size_t getNumSynapseDynamicsThreads(const SynapseGroupInternal&␣
↪sg);
    static void addPresynapticUpdateStrategy(PresynapticUpdateStrategy::Base*␣
↪strategy);
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::function<void(CodeStream&, Substitutions&)> Handler;
    typedef std::function<void(CodeStream&, const T&, Substitutions&)>␣
↪GroupHandler;
    typedef GroupHandler<NeuronGroupInternal> NeuronGroupHandler;
    typedef GroupHandler<SynapseGroupInternal> SynapseGroupHandler;
    typedef std::function<void(CodeStream&, const NeuronGroupInternal&,
↪ Substitutions&, NeuronGroupHandler, NeuronGroupHandler)>␣
↪NeuronGroupSimHandler;

    // methods

    BackendBase(int localHostID, const std::string& scalarType);
    virtual ~BackendBase();
    virtual void genNeuronUpdate(CodeStream& os, const ModelSpecInternal&␣
↪model, NeuronGroupSimHandler simHandler, NeuronGroupHandler␣
↪wuVarUpdateHandler) const = 0;

    virtual void genSynapseUpdate(
        CodeStream& os,
        const ModelSpecInternal& model,
        SynapseGroupHandler wumThreshHandler,
        SynapseGroupHandler wumSimHandler,
        SynapseGroupHandler wumEventHandler,
        SynapseGroupHandler postLearnHandler,
        SynapseGroupHandler synapseDynamicsHandler
        ) const = 0;

    virtual void genInit(
        CodeStream& os,
        const ModelSpecInternal& model,
        NeuronGroupHandler localNGHandler,
        NeuronGroupHandler remoteNGHandler,
        SynapseGroupHandler sgDenseInitHandler,
        SynapseGroupHandler sgSparseConnectHandler,
        SynapseGroupHandler sgSparseInitHandler
        ) const = 0;

    virtual void genDefinitionsPreamble(CodeStream& os) const = 0;
    virtual void genDefinitionsInternalPreamble(CodeStream& os) const = 0;
```

```
    virtual void genRunnerPreamble(CodeStream& os) const = 0;
    virtual void genAllocateMemPreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const = 0;
    virtual void genStepTimeFinalisePreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const = 0;

    virtual void genVariableDefinition(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual void genVariableImplementation(CodeStream& os, const std::string&␣
↪type, const std::string& name, VarLocation loc) const = 0;

    virtual MemAlloc genVariableAllocation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const = 0;

    virtual void genVariableFree(CodeStream& os, const std::string& name,␣
↪VarLocation loc) const = 0;

    virtual void genExtraGlobalParamDefinition(
        CodeStream& definitions,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual void genExtraGlobalParamImplementation(CodeStream& os, const␣
↪std::string& type, const std::string& name, VarLocation loc) const = 0;
    virtual void genExtraGlobalParamAllocation(CodeStream& os, const␣
↪std::string& type, const std::string& name, VarLocation loc) const = 0;
    virtual void genExtraGlobalParamPush(CodeStream& os, const std::string&␣
↪type, const std::string& name, VarLocation loc) const = 0;
    virtual void genExtraGlobalParamPull(CodeStream& os, const std::string&␣
↪type, const std::string& name, VarLocation loc) const = 0;
    virtual void genPopVariableInit(CodeStream& os, VarLocation loc, const␣
↪Substitutions& kernelSubs, Handler handler) const = 0;

    virtual void genVariableInit(
        CodeStream& os,
        VarLocation loc,
        size_t count,
        const std::string& indexVarName,
        const Substitutions& kernelSubs,
        Handler handler
        ) const = 0;
```

```
    virtual void genSynapseVariableRowInit(CodeStream& os, VarLocation loc,␣
↪const SynapseGroupInternal& sg, const Substitutions& kernelSubs, Handler␣
↪handler) const = 0;

    virtual void genVariablePush(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        bool autoInitialized,
        size_t count
        ) const = 0;

    virtual void genVariablePull(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const = 0;

    virtual void genCurrentTrueSpikePush(CodeStream& os, const␣
↪NeuronGroupInternal& ng) const = 0;
    virtual void genCurrentTrueSpikePull(CodeStream& os, const␣
↪NeuronGroupInternal& ng) const = 0;
    virtual void genCurrentSpikeLikeEventPush(CodeStream& os, const␣
↪NeuronGroupInternal& ng) const = 0;
    virtual void genCurrentSpikeLikeEventPull(CodeStream& os, const␣
↪NeuronGroupInternal& ng) const = 0;

    virtual MemAlloc genGlobalRNG(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const ModelSpecInternal& model
        ) const = 0;

    virtual MemAlloc genPopulationRNG(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const std::string& name,
        size_t count
        ) const = 0;

    virtual void genTimer(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
```

```
        CodeStream& allocations,
        CodeStream& free,
        CodeStream& stepTimeFinalise,
        const std::string& name,
        bool updateInStepTime
        ) const = 0;

    virtual void genMakefilePreamble(std::ostream& os) const = 0;
    virtual void genMakefileLinkRule(std::ostream& os) const = 0;
    virtual void genMakefileCompileRule(std::ostream& os) const = 0;
    virtual void genMSBuildConfigProperties(std::ostream& os) const = 0;
    virtual void genMSBuildImportProps(std::ostream& os) const = 0;
    virtual void genMSBuildItemDefinitions(std::ostream& os) const = 0;
    virtual void genMSBuildCompileModule(const std::string& moduleName,␣
↪std::ostream& os) const = 0;
    virtual void genMSBuildImportTarget(std::ostream& os) const = 0;
    virtual std::string getVarPrefix() const;
    virtual bool isGlobalRNGRequired(const ModelSpecInternal& model) const =␣
↪0;
    virtual bool isSynRemapRequired() const = 0;
    virtual bool isPostsynapticRemapRequired() const = 0;
    virtual size_t getDeviceMemoryBytes() const = 0;

    void genVariablePushPull(
        CodeStream& push,
        CodeStream& pull,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        bool autoInitialized,
        size_t count
        ) const;

    MemAlloc genArray(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const;

    void genScalar(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;
```

```
int getLocalHostID() const;
```

Detailed Documentation

Methods

```
virtual void genNeuronUpdate(
    CodeStream& os,
    const ModelSpecInternal& model,
    NeuronGroupSimHandler simHandler,
    NeuronGroupHandler wuVarUpdateHandler
    ) const
```

Generate platform-specific function to update the state of all neurons.

Parameters:

os	*CodeStream* to write function to
model	model to generate code for
simHandler	callback to write platform-independent code to update an individual *NeuronGroup*
wuVarUpdateHandler	callback to write platform-independent code to update pre and postsynaptic weight update model variables when neuron spikes

```
virtual void genSynapseUpdate(
    CodeStream& os,
    const ModelSpecInternal& model,
    SynapseGroupHandler wumThreshHandler,
    SynapseGroupHandler wumSimHandler,
    SynapseGroupHandler wumEventHandler,
    SynapseGroupHandler postLearnHandler,
    SynapseGroupHandler synapseDynamicsHandler
    ) const
```

Generate platform-specific function to update the state of all synapses.

Parameters:

os	*CodeStream* to write function to
model	model to generate code for
wumThreshHandler	callback to write platform-independent code to update an individual *NeuronGroup*
wumSimHandler	callback to write platform-independent code to process presynaptic spikes. "id_pre", "id_post" and "id_syn" variables; and either "addToInSynDelay" or "addToInSyn" function will be provided to callback via *Substitutions*.
wumEventHandler	callback to write platform-independent code to process presynaptic spike-like events. "id_pre", "id_post" and "id_syn" variables; and either "addToInSynDelay" or "addToInSyn" function will be provided to callback via *Substitutions*.
postLearnHandler	callback to write platform-independent code to process postsynaptic spikes. "id_pre", "id_post" and "id_syn" variables will be provided to callback via *Substitutions*.
synapseDynamicsHandler	callback to write platform-independent code to update time-driven synapse dynamics. "id_pre", "id_post" and "id_syn" variables; and either "addToInSynDelay" or "addToInSyn" function will be provided to callback via *Substitutions*.

```
virtual void genDefinitionsPreamble(CodeStream& os) const
```

Definitions is the usercode-facing header file for the generated code. This function generates a 'preamble' to this header file.

This will be included from a standard C++ compiler so shouldn't include any platform-specific types or headers

```
virtual void genDefinitionsInternalPreamble(CodeStream& os) const
```

Definitions internal is the internal header file for the generated code. This function generates a 'preamble' to this header file.

This will only be included by the platform-specific compiler used to build this backend so can include platform-specific types or headers

```
virtual void genAllocateMemPreamble(CodeStream& os, const ModelSpecInternal&␣
↪model) const
```

Allocate memory is the first function in GeNN generated code called by usercode and it should only ever be called once. Therefore it's a good place for any global initialisation. This function generates a 'preamble' to this function.

```
virtual void genStepTimeFinalisePreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const
```

After all timestep logic is complete.

```
virtual void genMakefilePreamble(std::ostream& os) const
```

This function can be used to generate a preamble for the GNU makefile used to build.

```
virtual void genMakefileLinkRule(std::ostream& os) const
```

The GNU make build system will populate a variable called '''' with a list of objects to link. This function should generate a GNU make rule to build these objects into a shared library.

```
virtual void genMakefileCompileRule(std::ostream& os) const
```

The GNU make build system uses 'pattern rules' (https://www.gnu.org/software/make/manual/html_node/Pattern-Intro.html) to build backend modules into objects. This function should generate a GNU make pattern rule capable of building each module (i.e. compiling .cc file $< into .o file $@).

```
virtual void genMSBuildConfigProperties(std::ostream& os) const
```

In MSBuild, 'properties' are used to configure global project settings e.g. whether the MSBuild project builds a static or dynamic library This function can be used to add additional XML properties to this section.

see https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-properties for more information.

```
virtual void genMSBuildItemDefinitions(std::ostream& os) const
```

In MSBuild, the 'item definitions' are used to override the default properties of 'items' such as `<ClCompile>` or `<Link>`. This function should generate XML to correctly configure the 'items' required to build the generated code, taking into account '''' etc.

see https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-items#item-definitions for more information.

```
virtual std::string getVarPrefix() const
```

When backends require separate 'device' and 'host' versions of variables, they are identified with a prefix. This function returns this prefix so it can be used in otherwise platform-independent code.

```
virtual bool isGlobalRNGRequired(const ModelSpecInternal& model) const
```

Different backends use different RNGs for different things. Does this one require a global RNG for the specified model?

```
virtual size_t getDeviceMemoryBytes() const
```

How many bytes of memory does 'device' have.

```
static void addPresynapticUpdateStrategy(PresynapticUpdateStrategy::Base*␣
↪strategy)
```

Register a new presynaptic update strategy.

This function should be called with strategies in ascending order of preference

Overview

```
namespace CUDA

// namespaces

namespace CodeGenerator::CUDA::Optimiser;
namespace CodeGenerator::CUDA::PresynapticUpdateStrategy;
namespace CodeGenerator::CUDA::Utils;

// typedefs

typedef std::array<size_t, KernelMax> KernelBlockSize;

// enums

enum BlockSizeSelect;
enum DeviceSelect;
enum Kernel;

// structs

struct Preferences;

// classes

class Backend;

 // namespace CUDA
```

Detailed Documentation

Typedefs

```
typedef std::array<size_t, KernelMax> KernelBlockSize
```

Array of block sizes for each kernel.

28.1.2 namespace CodeGenerator::SingleThreadedCPU

namespace CodeGenerator::SingleThreadedCPU::Optimiser

```
namespace Optimiser
```

```
// global functions

BACKEND_EXPORT Backend createBackend(
    const ModelSpecInternal& model,
    const filesystem::path& outputPath,
    int localHostID,
    const Preferences& preferences
    );

 // namespace Optimiser
```

struct CodeGenerator::SingleThreadedCPU::Preferences

```
#include <backend.h>

struct Preferences: public CodeGenerator::PreferencesBase

;
```

Inherited Members

```
public:
    // fields

    bool optimizeCode;
    bool debugCode;
    std::string userCxxFlagsGNU;
    std::string userNvccFlagsGNU;
    plog::Severity logLevel;
```

class CodeGenerator::SingleThreadedCPU::Backend

Overview

```
#include <backend.h>

class Backend: public CodeGenerator::BackendBase

public:
    // methods

    Backend(
        int localHostID,
        const std::string& scalarType,
        const Preferences& preferences
        );

    virtual void genNeuronUpdate(CodeStream& os, const ModelSpecInternal&␣
↪model, NeuronGroupSimHandler simHandler, NeuronGroupHandler␣
↪wuVarUpdateHandler) const;
```

```
    virtual void genSynapseUpdate(
        CodeStream& os,
        const ModelSpecInternal& model,
        SynapseGroupHandler wumThreshHandler,
        SynapseGroupHandler wumSimHandler,
        SynapseGroupHandler wumEventHandler,
        SynapseGroupHandler postLearnHandler,
        SynapseGroupHandler synapseDynamicsHandler
        ) const;

    virtual void genInit(
        CodeStream& os,
        const ModelSpecInternal& model,
        NeuronGroupHandler localNGHandler,
        NeuronGroupHandler remoteNGHandler,
        SynapseGroupHandler sgDenseInitHandler,
        SynapseGroupHandler sgSparseConnectHandler,
        SynapseGroupHandler sgSparseInitHandler
        ) const;

    virtual void genDefinitionsPreamble(CodeStream& os) const;
    virtual void genDefinitionsInternalPreamble(CodeStream& os) const;
    virtual void genRunnerPreamble(CodeStream& os) const;
    virtual void genAllocateMemPreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const;
    virtual void genStepTimeFinalisePreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const;

    virtual void genVariableDefinition(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    virtual void genVariableImplementation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    virtual MemAlloc genVariableAllocation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const;

    virtual void genVariableFree(
        CodeStream& os,
        const std::string& name,
```

```
    VarLocation loc
    ) const;

virtual void genExtraGlobalParamDefinition(
    CodeStream& definitions,
    const std::string& type,
    const std::string& name,
    VarLocation loc
    ) const;

virtual void genExtraGlobalParamImplementation(
    CodeStream& os,
    const std::string& type,
    const std::string& name,
    VarLocation loc
    ) const;

virtual void genExtraGlobalParamAllocation(
    CodeStream& os,
    const std::string& type,
    const std::string& name,
    VarLocation loc
    ) const;

virtual void genExtraGlobalParamPush(
    CodeStream& os,
    const std::string& type,
    const std::string& name,
    VarLocation loc
    ) const;

virtual void genExtraGlobalParamPull(
    CodeStream& os,
    const std::string& type,
    const std::string& name,
    VarLocation loc
    ) const;

virtual void genPopVariableInit(
    CodeStream& os,
    VarLocation loc,
    const Substitutions& kernelSubs,
    Handler handler
    ) const;

virtual void genVariableInit(
    CodeStream& os,
    VarLocation loc,
    size_t count,
    const std::string& indexVarName,
    const Substitutions& kernelSubs,
    Handler handler
    ) const;
```

```
    virtual void genSynapseVariableRowInit(
        CodeStream& os,
        VarLocation loc,
        const SynapseGroupInternal& sg,
        const Substitutions& kernelSubs,
        Handler handler
        ) const;

    virtual void genCurrentTrueSpikePush(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const;

    virtual void genCurrentTrueSpikePull(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const;

    virtual void genCurrentSpikeLikeEventPush(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const;

    virtual void genCurrentSpikeLikeEventPull(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const;

    virtual void genVariablePush(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        bool autoInitialized,
        size_t count
        ) const;

    virtual void genVariablePull(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const;

    virtual MemAlloc genGlobalRNG(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const ModelSpecInternal& model
        ) const;
```

```
    virtual MemAlloc genPopulationRNG(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const std::string& name,
        size_t count
        ) const;

    virtual void genTimer(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        CodeStream& stepTimeFinalise,
        const std::string& name,
        bool updateInStepTime
        ) const;

    virtual void genMakefilePreamble(std::ostream& os) const;
    virtual void genMakefileLinkRule(std::ostream& os) const;
    virtual void genMakefileCompileRule(std::ostream& os) const;
    virtual void genMSBuildConfigProperties(std::ostream& os) const;
    virtual void genMSBuildImportProps(std::ostream& os) const;
    virtual void genMSBuildItemDefinitions(std::ostream& os) const;

    virtual void genMSBuildCompileModule(
        const std::string& moduleName,
        std::ostream& os
        ) const;

    virtual void genMSBuildImportTarget(std::ostream& os) const;
    virtual std::string getVarPrefix() const;
    virtual bool isGlobalRNGRequired(const ModelSpecInternal& model) const;
    virtual bool isSynRemapRequired() const;
    virtual bool isPostsynapticRemapRequired() const;
    virtual size_t getDeviceMemoryBytes() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::function<void(CodeStream&, Substitutions&)> Handler;
    typedef std::function<void(CodeStream&, const T&, Substitutions&)>␣
↪GroupHandler;
    typedef GroupHandler<NeuronGroupInternal> NeuronGroupHandler;
    typedef GroupHandler<SynapseGroupInternal> SynapseGroupHandler;
    typedef std::function<void(CodeStream&, const NeuronGroupInternal&,
↪ Substitutions&, NeuronGroupHandler, NeuronGroupHandler)>␣
```

```
↪NeuronGroupSimHandler;

    // methods

    BackendBase(int localHostID, const std::string& scalarType);
    virtual ~BackendBase();
    virtual void genNeuronUpdate(CodeStream& os, const ModelSpecInternal&␣
↪model, NeuronGroupSimHandler simHandler, NeuronGroupHandler␣
↪wuVarUpdateHandler) const = 0;

    virtual void genSynapseUpdate(
        CodeStream& os,
        const ModelSpecInternal& model,
        SynapseGroupHandler wumThreshHandler,
        SynapseGroupHandler wumSimHandler,
        SynapseGroupHandler wumEventHandler,
        SynapseGroupHandler postLearnHandler,
        SynapseGroupHandler synapseDynamicsHandler
        ) const = 0;

    virtual void genInit(
        CodeStream& os,
        const ModelSpecInternal& model,
        NeuronGroupHandler localNGHandler,
        NeuronGroupHandler remoteNGHandler,
        SynapseGroupHandler sgDenseInitHandler,
        SynapseGroupHandler sgSparseConnectHandler,
        SynapseGroupHandler sgSparseInitHandler
        ) const = 0;

    virtual void genDefinitionsPreamble(CodeStream& os) const = 0;
    virtual void genDefinitionsInternalPreamble(CodeStream& os) const = 0;
    virtual void genRunnerPreamble(CodeStream& os) const = 0;
    virtual void genAllocateMemPreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const = 0;
    virtual void genStepTimeFinalisePreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const = 0;

    virtual void genVariableDefinition(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual void genVariableImplementation(CodeStream& os, const std::string&␣
↪type, const std::string& name, VarLocation loc) const = 0;

    virtual MemAlloc genVariableAllocation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
```

```
        size_t count
        ) const = 0;

    virtual void genVariableFree(CodeStream& os, const std::string& name,␣
↪VarLocation loc) const = 0;

    virtual void genExtraGlobalParamDefinition(
        CodeStream& definitions,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual void genExtraGlobalParamImplementation(CodeStream& os, const␣
↪std::string& type, const std::string& name, VarLocation loc) const = 0;
    virtual void genExtraGlobalParamAllocation(CodeStream& os, const␣
↪std::string& type, const std::string& name, VarLocation loc) const = 0;
    virtual void genExtraGlobalParamPush(CodeStream& os, const std::string&␣
↪type, const std::string& name, VarLocation loc) const = 0;
    virtual void genExtraGlobalParamPull(CodeStream& os, const std::string&␣
↪type, const std::string& name, VarLocation loc) const = 0;
    virtual void genPopVariableInit(CodeStream& os, VarLocation loc, const␣
↪Substitutions& kernelSubs, Handler handler) const = 0;

    virtual void genVariableInit(
        CodeStream& os,
        VarLocation loc,
        size_t count,
        const std::string& indexVarName,
        const Substitutions& kernelSubs,
        Handler handler
        ) const = 0;

    virtual void genSynapseVariableRowInit(CodeStream& os, VarLocation loc,␣
↪const SynapseGroupInternal& sg, const Substitutions& kernelSubs, Handler␣
↪handler) const = 0;

    virtual void genVariablePush(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        bool autoInitialized,
        size_t count
        ) const = 0;

    virtual void genVariablePull(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const = 0;
```

```
    virtual void genCurrentTrueSpikePush(CodeStream& os, const␣
↪NeuronGroupInternal& ng) const = 0;
    virtual void genCurrentTrueSpikePull(CodeStream& os, const␣
↪NeuronGroupInternal& ng) const = 0;
    virtual void genCurrentSpikeLikeEventPush(CodeStream& os, const␣
↪NeuronGroupInternal& ng) const = 0;
    virtual void genCurrentSpikeLikeEventPull(CodeStream& os, const␣
↪NeuronGroupInternal& ng) const = 0;

    virtual MemAlloc genGlobalRNG(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const ModelSpecInternal& model
        ) const = 0;

    virtual MemAlloc genPopulationRNG(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const std::string& name,
        size_t count
        ) const = 0;

    virtual void genTimer(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        CodeStream& stepTimeFinalise,
        const std::string& name,
        bool updateInStepTime
        ) const = 0;

    virtual void genMakefilePreamble(std::ostream& os) const = 0;
    virtual void genMakefileLinkRule(std::ostream& os) const = 0;
    virtual void genMakefileCompileRule(std::ostream& os) const = 0;
    virtual void genMSBuildConfigProperties(std::ostream& os) const = 0;
    virtual void genMSBuildImportProps(std::ostream& os) const = 0;
    virtual void genMSBuildItemDefinitions(std::ostream& os) const = 0;
    virtual void genMSBuildCompileModule(const std::string& moduleName,␣
↪std::ostream& os) const = 0;
    virtual void genMSBuildImportTarget(std::ostream& os) const = 0;
    virtual std::string getVarPrefix() const;
    virtual bool isGlobalRNGRequired(const ModelSpecInternal& model) const =␣
↪0;
    virtual bool isSynRemapRequired() const = 0;
    virtual bool isPostsynapticRemapRequired() const = 0;
    virtual size_t getDeviceMemoryBytes() const = 0;
```

```
    void genVariablePushPull(
        CodeStream& push,
        CodeStream& pull,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        bool autoInitialized,
        size_t count
        ) const;

    MemAlloc genArray(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const;

    void genScalar(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    int getLocalHostID() const;
```

Detailed Documentation

Methods

```
virtual void genNeuronUpdate(
    CodeStream& os,
    const ModelSpecInternal& model,
    NeuronGroupSimHandler simHandler,
    NeuronGroupHandler wuVarUpdateHandler
    ) const
```

Generate platform-specific function to update the state of all neurons.

Parameters:

os	*CodeStream* to write function to
model	model to generate code for
simHandler	callback to write platform-independent code to update an individual *NeuronGroup*
wuVarUpdateHandler	callback to write platform-independent code to update pre and postsynaptic weight update model variables when neuron spikes

```
virtual void genSynapseUpdate(
    CodeStream& os,
    const ModelSpecInternal& model,
    SynapseGroupHandler wumThreshHandler,
    SynapseGroupHandler wumSimHandler,
    SynapseGroupHandler wumEventHandler,
    SynapseGroupHandler postLearnHandler,
    SynapseGroupHandler synapseDynamicsHandler
    ) const
```

Generate platform-specific function to update the state of all synapses.

Parameters:

os	*CodeStream* to write function to
model	model to generate code for
wumThreshHandler	callback to write platform-independent code to update an individual *NeuronGroup*
wumSimHandler	callback to write platform-independent code to process presynaptic spikes. "id_pre", "id_post" and "id_syn" variables; and either "addToInSynDelay" or "addToInSyn" function will be provided to callback via *Substitutions*.
wumEventHandler	callback to write platform-independent code to process presynaptic spike-like events. "id_pre", "id_post" and "id_syn" variables; and either "addToInSynDelay" or "addToInSyn" function will be provided to callback via *Substitutions*.
postLearnHandler	callback to write platform-independent code to process postsynaptic spikes. "id_pre", "id_post" and "id_syn" variables will be provided to callback via *Substitutions*.
synapseDynamicsHandler	callback to write platform-independent code to update time-driven synapse dynamics. "id_pre", "id_post" and "id_syn" variables; and either "addToInSynDelay" or "addToInSyn" function will be provided to callback via *Substitutions*.

```
virtual void genDefinitionsPreamble(CodeStream& os) const
```

Definitions is the usercode-facing header file for the generated code. This function generates a 'preamble' to this header file.

This will be included from a standard C++ compiler so shouldn't include any platform-specific types or headers

```
virtual void genDefinitionsInternalPreamble(CodeStream& os) const
```

Definitions internal is the internal header file for the generated code. This function generates a 'preamble' to this header file.

This will only be included by the platform-specific compiler used to build this backend so can include platform-specific types or headers

```
virtual void genAllocateMemPreamble(CodeStream& os, const ModelSpecInternal&␣
↪model) const
```

Allocate memory is the first function in GeNN generated code called by usercode and it should only ever be called once. Therefore it's a good place for any global initialisation. This function generates a 'preamble' to this function.

```
virtual void genStepTimeFinalisePreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const
```

After all timestep logic is complete.

```
virtual void genMakefilePreamble(std::ostream& os) const
```

This function can be used to generate a preamble for the GNU makefile used to build.

```
virtual void genMakefileLinkRule(std::ostream& os) const
```

The GNU make build system will populate a variable called ‘‘‘‘ with a list of objects to link. This function should generate a GNU make rule to build these objects into a shared library.

```
virtual void genMakefileCompileRule(std::ostream& os) const
```

The GNU make build system uses 'pattern rules' (https://www.gnu.org/software/make/manual/html_node/Pattern-Intro.html) to build backend modules into objects. This function should generate a GNU make pattern rule capable of building each module (i.e. compiling .cc file $< into .o file $@).

```
virtual void genMSBuildConfigProperties(std::ostream& os) const
```

In MSBuild, 'properties' are used to configure global project settings e.g. whether the MSBuild project builds a static or dynamic library This function can be used to add additional XML properties to this section.

see https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-properties for more information.

```
virtual void genMSBuildItemDefinitions(std::ostream& os) const
```

In MSBuild, the 'item definitions' are used to override the default properties of 'items' such as `<ClCompile>` or `<Link>`. This function should generate XML to correctly configure the 'items' required to build the generated code, taking into account ‘‘‘‘ etc.

see https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-items#item-definitions for more information.

```
virtual std::string getVarPrefix() const
```

When backends require separate 'device' and 'host' versions of variables, they are identified with a prefix. This function returns this prefix so it can be used in otherwise platform-independent code.

```
virtual bool isGlobalRNGRequired(const ModelSpecInternal& model) const
```

Different backends use different RNGs for different things. Does this one require a global RNG for the specified model?

```
virtual size_t getDeviceMemoryBytes() const
```

How many bytes of memory does 'device' have.

```
namespace SingleThreadedCPU

// namespaces

namespace CodeGenerator::SingleThreadedCPU::Optimiser;

// structs

struct Preferences;

// classes
```

```
class Backend;

 // namespace SingleThreadedCPU
```

28.1.3 struct CodeGenerator::FunctionTemplate

Overview

Immutable structure for specifying how to implement a generic function e.g. *More...*

```
#include <codeGenUtils.h>

struct FunctionTemplate

    // fields

    const std::string genericName;
    const unsigned int numArguments;
    const std::string doublePrecisionTemplate;
    const std::string singlePrecisionTemplate;

    // methods

    FunctionTemplate operator = (const FunctionTemplate& o);
;
```

Detailed Documentation

Immutable structure for specifying how to implement a generic function e.g. gennrand_uniform

NOTE for the sake of easy initialisation first two parameters of GenericFunction are repeated (C++17 fixes)

Fields

`const std::string genericName`

Generic name used to refer to function in user code.

`const unsigned int numArguments`

Number of function arguments.

`const std::string doublePrecisionTemplate`

The function template (for use with *functionSubstitute*) used when model uses double precision.

`const std::string singlePrecisionTemplate`

The function template (for use with *functionSubstitute*) used when model uses single precision.

28.1.4 template struct CodeGenerator::NameIterCtx

```
#include <codeGenUtils.h>

template <typename Container>
```

```
struct NameIterCtx

    // typedefs

    typedef StructNameConstIter<typename Container::const_iterator> NameIter;

    // fields

    const Container container;
    const NameIter nameBegin;
    const NameIter nameEnd;

    // methods

    NameIterCtx(const Container& c);
;
```

28.1.5 struct CodeGenerator::PreferencesBase

Overview

Base class for backend preferences - can be accessed via a global in 'classic' C++ code generator. *More...*

```
#include <backendBase.h>

struct PreferencesBase

    // fields

    bool optimizeCode;
    bool debugCode;
    std::string userCxxFlagsGNU;
    std::string userNvccFlagsGNU;
    plog::Severity logLevel;
;

// direct descendants

struct Preferences;
struct Preferences;
```

Detailed Documentation

Base class for backend preferences - can be accessed via a global in 'classic' C++ code generator.

Fields

`bool optimizeCode`

Generate speed-optimized code, potentially at the expense of floating-point accuracy.

`bool debugCode`

Generate code with debug symbols.

`std::string userCxxFlagsGNU`

C++ compiler options to be used for building all host side code (used for unix based platforms)

`std::string userNvccFlagsGNU`

NVCC compiler options they may want to use for all GPU code (used for unix based platforms)

`plog::Severity logLevel`

Logging level to use for code generation.

28.1.6 class CodeGenerator::BackendBase

Overview

```
#include <backendBase.h>

class BackendBase

public:
    // typedefs

    typedef std::function<void(CodeStream&, Substitutions&)> Handler;
    typedef std::function<void(CodeStream&, const T&, Substitutions&)>␣
↪GroupHandler;
    typedef GroupHandler<NeuronGroupInternal> NeuronGroupHandler;
    typedef GroupHandler<SynapseGroupInternal> SynapseGroupHandler;
    typedef std::function<void(CodeStream&, const NeuronGroupInternal&,
↪ Substitutions&, NeuronGroupHandler, NeuronGroupHandler)>␣
↪NeuronGroupSimHandler;

    // methods

    BackendBase(
        int localHostID,
        const std::string& scalarType
        );

    virtual ~BackendBase();
    virtual void genNeuronUpdate(CodeStream& os, const ModelSpecInternal&␣
↪model, NeuronGroupSimHandler simHandler, NeuronGroupHandler␣
↪wuVarUpdateHandler) const = 0;

    virtual void genSynapseUpdate(
        CodeStream& os,
        const ModelSpecInternal& model,
        SynapseGroupHandler wumThreshHandler,
        SynapseGroupHandler wumSimHandler,
        SynapseGroupHandler wumEventHandler,
        SynapseGroupHandler postLearnHandler,
        SynapseGroupHandler synapseDynamicsHandler
        ) const = 0;
```

```
    virtual void genInit(
        CodeStream& os,
        const ModelSpecInternal& model,
        NeuronGroupHandler localNGHandler,
        NeuronGroupHandler remoteNGHandler,
        SynapseGroupHandler sgDenseInitHandler,
        SynapseGroupHandler sgSparseConnectHandler,
        SynapseGroupHandler sgSparseInitHandler
        ) const = 0;

    virtual void genDefinitionsPreamble(CodeStream& os) const = 0;
    virtual void genDefinitionsInternalPreamble(CodeStream& os) const = 0;
    virtual void genRunnerPreamble(CodeStream& os) const = 0;
    virtual void genAllocateMemPreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const = 0;
    virtual void genStepTimeFinalisePreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const = 0;

    virtual void genVariableDefinition(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual void genVariableImplementation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual MemAlloc genVariableAllocation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const = 0;

    virtual void genVariableFree(
        CodeStream& os,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual void genExtraGlobalParamDefinition(
        CodeStream& definitions,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;
```

```
    virtual void genExtraGlobalParamImplementation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual void genExtraGlobalParamAllocation(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual void genExtraGlobalParamPush(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual void genExtraGlobalParamPull(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const = 0;

    virtual void genPopVariableInit(
        CodeStream& os,
        VarLocation loc,
        const Substitutions& kernelSubs,
        Handler handler
        ) const = 0;

    virtual void genVariableInit(
        CodeStream& os,
        VarLocation loc,
        size_t count,
        const std::string& indexVarName,
        const Substitutions& kernelSubs,
        Handler handler
        ) const = 0;

    virtual void genSynapseVariableRowInit(
        CodeStream& os,
        VarLocation loc,
        const SynapseGroupInternal& sg,
        const Substitutions& kernelSubs,
        Handler handler
        ) const = 0;

    virtual void genVariablePush(
        CodeStream& os,
```

```
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        bool autoInitialized,
        size_t count
        ) const = 0;

    virtual void genVariablePull(
        CodeStream& os,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const = 0;

    virtual void genCurrentTrueSpikePush(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const = 0;

    virtual void genCurrentTrueSpikePull(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const = 0;

    virtual void genCurrentSpikeLikeEventPush(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const = 0;

    virtual void genCurrentSpikeLikeEventPull(
        CodeStream& os,
        const NeuronGroupInternal& ng
        ) const = 0;

    virtual MemAlloc genGlobalRNG(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const ModelSpecInternal& model
        ) const = 0;

    virtual MemAlloc genPopulationRNG(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const std::string& name,
        size_t count
        ) const = 0;
```

```
    virtual void genTimer(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        CodeStream& stepTimeFinalise,
        const std::string& name,
        bool updateInStepTime
        ) const = 0;

    virtual void genMakefilePreamble(std::ostream& os) const = 0;
    virtual void genMakefileLinkRule(std::ostream& os) const = 0;
    virtual void genMakefileCompileRule(std::ostream& os) const = 0;
    virtual void genMSBuildConfigProperties(std::ostream& os) const = 0;
    virtual void genMSBuildImportProps(std::ostream& os) const = 0;
    virtual void genMSBuildItemDefinitions(std::ostream& os) const = 0;

    virtual void genMSBuildCompileModule(
        const std::string& moduleName,
        std::ostream& os
        ) const = 0;

    virtual void genMSBuildImportTarget(std::ostream& os) const = 0;
    virtual std::string getVarPrefix() const;
    virtual bool isGlobalRNGRequired(const ModelSpecInternal& model) const =␣
↪0;
    virtual bool isSynRemapRequired() const = 0;
    virtual bool isPostsynapticRemapRequired() const = 0;
    virtual size_t getDeviceMemoryBytes() const = 0;

    void genVariablePushPull(
        CodeStream& push,
        CodeStream& pull,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        bool autoInitialized,
        size_t count
        ) const;

    MemAlloc genArray(
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        CodeStream& allocations,
        CodeStream& free,
        const std::string& type,
        const std::string& name,
        VarLocation loc,
        size_t count
        ) const;

    void genScalar(
```

```
        CodeStream& definitions,
        CodeStream& definitionsInternal,
        CodeStream& runner,
        const std::string& type,
        const std::string& name,
        VarLocation loc
        ) const;

    int getLocalHostID() const;
;

// direct descendants

class Backend;
class Backend;
```

Detailed Documentation

Typedefs

```
typedef GroupHandler<NeuronGroupInternal> NeuronGroupHandler
```

Standard callback type which provides a *CodeStream* to write platform-independent code for the specified *NeuronGroup* to.

```
typedef GroupHandler<SynapseGroupInternal> SynapseGroupHandler
```

Standard callback type which provides a *CodeStream* to write platform-independent code for the specified *SynapseGroup* to.

```
typedef std::function<void(CodeStream&, const NeuronGroupInternal&,
↪ Substitutions&, NeuronGroupHandler, NeuronGroupHandler)>␣
↪NeuronGroupSimHandler
```

Callback function type for generation neuron group simulation code.

Provides additional callbacks to insert code to emit spikes

Methods

```
virtual void genNeuronUpdate(
    CodeStream& os,
    const ModelSpecInternal& model,
    NeuronGroupSimHandler simHandler,
    NeuronGroupHandler wuVarUpdateHandler
    ) const = 0
```

Generate platform-specific function to update the state of all neurons.

Parameters:

os	*CodeStream* to write function to
model	model to generate code for
simHandler	callback to write platform-independent code to update an individual *NeuronGroup*
wuVarUpdateHandler	callback to write platform-independent code to update pre and postsynaptic weight update model variables when neuron spikes

```
virtual void genSynapseUpdate(
    CodeStream& os,
    const ModelSpecInternal& model,
    SynapseGroupHandler wumThreshHandler,
    SynapseGroupHandler wumSimHandler,
    SynapseGroupHandler wumEventHandler,
    SynapseGroupHandler postLearnHandler,
    SynapseGroupHandler synapseDynamicsHandler
    ) const = 0
```

Generate platform-specific function to update the state of all synapses.

Parameters:

os	*CodeStream* to write function to
model	model to generate code for
wumThreshHandler	callback to write platform-independent code to update an individual *NeuronGroup*
wumSimHandler	callback to write platform-independent code to process presynaptic spikes. "id_pre", "id_post" and "id_syn" variables; and either "addToInSynDelay" or "addToInSyn" function will be provided to callback via *Substitutions*.
wumEventHandler	callback to write platform-independent code to process presynaptic spike-like events. "id_pre", "id_post" and "id_syn" variables; and either "addToInSynDelay" or "addToInSyn" function will be provided to callback via *Substitutions*.
postLearnHandler	callback to write platform-independent code to process postsynaptic spikes. "id_pre", "id_post" and "id_syn" variables will be provided to callback via *Substitutions*.
synapseDynamicsHandler	callback to write platform-independent code to update time-driven synapse dynamics. "id_pre", "id_post" and "id_syn" variables; and either "addToInSynDelay" or "addToInSyn" function will be provided to callback via *Substitutions*.

```
virtual void genDefinitionsPreamble(CodeStream& os) const = 0
```

Definitions is the usercode-facing header file for the generated code. This function generates a 'preamble' to this header file.

This will be included from a standard C++ compiler so shouldn't include any platform-specific types or headers

```
virtual void genDefinitionsInternalPreamble(CodeStream& os) const = 0
```

Definitions internal is the internal header file for the generated code. This function generates a 'preamble' to this header file.

This will only be included by the platform-specific compiler used to build this backend so can include platform-specific types or headers

```
virtual void genAllocateMemPreamble(CodeStream& os, const ModelSpecInternal&␣
↪model) const = 0
```

Allocate memory is the first function in GeNN generated code called by usercode and it should only ever be called once. Therefore it's a good place for any global initialisation. This function generates a 'preamble' to this function.

```
virtual void genStepTimeFinalisePreamble(CodeStream& os, const␣
↪ModelSpecInternal& model) const = 0
```

After all timestep logic is complete.

```
virtual void genMakefilePreamble(std::ostream& os) const = 0
```

This function can be used to generate a preamble for the GNU makefile used to build.

```
virtual void genMakefileLinkRule(std::ostream& os) const = 0
```

The GNU make build system will populate a variable called '''' with a list of objects to link. This function should generate a GNU make rule to build these objects into a shared library.

```
virtual void genMakefileCompileRule(std::ostream& os) const = 0
```

The GNU make build system uses 'pattern rules' (https://www.gnu.org/software/make/manual/html_node/Pattern-Intro.html) to build backend modules into objects. This function should generate a GNU make pattern rule capable of building each module (i.e. compiling .cc file $< into .o file $@).

```
virtual void genMSBuildConfigProperties(std::ostream& os) const = 0
```

In MSBuild, 'properties' are used to configure global project settings e.g. whether the MSBuild project builds a static or dynamic library This function can be used to add additional XML properties to this section.

see https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-properties for more information.

```
virtual void genMSBuildItemDefinitions(std::ostream& os) const = 0
```

In MSBuild, the 'item definitions' are used to override the default properties of 'items' such as `<ClCompile>` or `<Link>`. This function should generate XML to correctly configure the 'items' required to build the generated code, taking into account '''' etc.

see https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-items#item-definitions for more information.

```
virtual std::string getVarPrefix() const
```

When backends require separate 'device' and 'host' versions of variables, they are identified with a prefix. This function returns this prefix so it can be used in otherwise platform-independent code.

```
virtual bool isGlobalRNGRequired(const ModelSpecInternal& model) const = 0
```

Different backends use different RNGs for different things. Does this one require a global RNG for the specified model?

```
virtual size_t getDeviceMemoryBytes() const = 0
```

How many bytes of memory does 'device' have.

```
void genVariablePushPull(
    CodeStream& push,
    CodeStream& pull,
    const std::string& type,
    const std::string& name,
    VarLocation loc,
    bool autoInitialized,
    size_t count
    ) const
```

Helper function to generate matching push and pull functions for a variable.

```
MemAlloc genArray(
    CodeStream& definitions,
    CodeStream& definitionsInternal,
    CodeStream& runner,
    CodeStream& allocations,
    CodeStream& free,
    const std::string& type,
    const std::string& name,
    VarLocation loc,
    size_t count
    ) const
```

Helper function to generate matching definition, declaration, allocation and free code for an array.

```
void genScalar(
    CodeStream& definitions,
    CodeStream& definitionsInternal,
    CodeStream& runner,
    const std::string& type,
    const std::string& name,
    VarLocation loc
    ) const
```

Helper function to generate matching definition and declaration code for a scalar variable.

```
int getLocalHostID() const
```

Gets ID of local host backend is building code for.

28.1.7 class CodeGenerator::CodeStream

struct CodeGenerator::CodeStream::CB

Overview

A close bracket marker. *More...*

```
#include <codeStream.h>

struct CB

    // fields

    const unsigned int Level;

    // methods

    CB(unsigned int level);
;
```

Detailed Documentation

A close bracket marker.

Write to code stream `os` using:

```
os << CB(16);
```

struct CodeGenerator::CodeStream::OB

Overview

An open bracket marker. *More...*

```
#include <codeStream.h>

struct OB

    // fields

    const unsigned int Level;

    // methods

    OB(unsigned int level);
;
```

Detailed Documentation

An open bracket marker.

Write to code stream `os` using:

```
os << OB(16);
```

class CodeGenerator::CodeStream::IndentBuffer

```
class IndentBuffer: public streambuf

public:
    // methods

    IndentBuffer();
    void indent();
    void deindent();
    void setSink(std::streambuf* sink);
;
```

class CodeGenerator::CodeStream::Scope

```
#include <codeStream.h>

class Scope

public:
    // methods

    Scope(CodeStream& codeStream);
```

```
    ~Scope();
;
#include <codeStream.h>

class CodeStream: public std::ostream

public:
    // structs

    struct CB;
    struct OB;

    // classes

    class IndentBuffer;
    class Scope;

    // methods

    CodeStream();
    CodeStream(std::ostream& stream);
    void setSink(std::ostream& stream);
;
```

28.1.8 class CodeGenerator::MemAlloc

```
#include <backendBase.h>

class MemAlloc

public:
    // methods

    size_t getHostBytes() const;
    size_t getDeviceBytes() const;
    size_t getZeroCopyBytes() const;
    size_t getHostMBytes() const;
    size_t getDeviceMBytes() const;
    size_t getZeroCopyMBytes() const;
    MemAlloc& operator += (const MemAlloc& rhs);
    static MemAlloc zero();
    static MemAlloc host(size_t hostBytes);
    static MemAlloc device(size_t deviceBytes);
    static MemAlloc zeroCopy(size_t zeroCopyBytes);
;
```

28.1.9 template class CodeGenerator::StructNameConstIter

Custom iterator for iterating through the containers of structs with 'name' members.

```
#include <codeGenUtils.h>
```

```
template <typename BaseIter>
class StructNameConstIter: public BaseIter

public:
    // methods

    StructNameConstIter();
    StructNameConstIter(BaseIter iter);
    const std::string* operator -> () const;
    const std::string& operator * () const;
;
```

28.1.10 class CodeGenerator::Substitutions

```
#include <substitutions.h>

class Substitutions

public:
    // methods

    Substitutions(const Substitutions* parent = nullptr);

    Substitutions(
        const std::vector<FunctionTemplate>& functions,
        const std::string& ftype
        );

    void addVarSubstitution(
        const std::string& source,
        const std::string& destionation,
        bool allowOverride = false
        );

    void addFuncSubstitution(
        const std::string& source,
        unsigned int numArguments,
        const std::string& funcTemplate,
        bool allowOverride = false
        );

    bool hasVarSubstitution(const std::string& source) const;
    const std::string& getVarSubstitution(const std::string& source) const;
    void apply(std::string& code) const;
    const std::string operator [] (const std::string& source) const;
;
```

28.1.11 class CodeGenerator::TeeBuf

```
#include <teeStream.h>

class TeeBuf: public streambuf
```

```
public:
    // methods

    template <typename... T>
    TeeBuf(T&&... streamBufs);
;
```

28.1.12 class CodeGenerator::TeeStream

```
#include <teeStream.h>

class TeeStream: public std::ostream

public:
    // methods

    template <typename... T>
    TeeStream(T&&... streamBufs);
;
```

28.1.13 Overview

Helper class for generating code - automatically inserts brackets, indents etc. *More...*

```
namespace CodeGenerator

// namespaces

namespace CodeGenerator::CUDA;
    namespace CodeGenerator::CUDA::Optimiser;
    namespace CodeGenerator::CUDA::PresynapticUpdateStrategy;
    namespace CodeGenerator::CUDA::Utils;
namespace CodeGenerator::SingleThreadedCPU;
    namespace CodeGenerator::SingleThreadedCPU::Optimiser;

// typedefs

typedef NameIterCtx<Models::Base::VarVec> VarNameIterCtx;
typedef NameIterCtx<Snippet::Base::EGPVec> EGPNameIterCtx;
typedef NameIterCtx<Snippet::Base::DerivedParamVec> DerivedParamNameIterCtx;
typedef NameIterCtx<Snippet::Base::ParamValVec> ParamValIterCtx;

// structs

struct FunctionTemplate;

template <typename Container>
struct NameIterCtx;

struct PreferencesBase;

// classes
```

```
class BackendBase;
class CodeStream;
class MemAlloc;

template <typename BaseIter>
class StructNameConstIter;

class Substitutions;
class TeeBuf;
class TeeStream;

// global functions

void substitute(std::string& s, const std::string& trg, const std::string&␣
↪rep);
bool regexVarSubstitute(std::string& s, const std::string& trg, const␣
↪std::string& rep);
bool regexFuncSubstitute(std::string& s, const std::string& trg, const␣
↪std::string& rep);

void functionSubstitute(
    std::string& code,
    const std::string& funcName,
    unsigned int numParams,
    const std::string& replaceFuncTemplate
    );

template <typename NameIter>
void name_substitutions(
    std::string& code,
    const std::string& prefix,
    NameIter namesBegin,
    NameIter namesEnd,
    const std::string& postfix = "",
    const std::string& ext = ""
    );

void name_substitutions(
    std::string& code,
    const std::string& prefix,
    const std::vector<std::string>& names,
    const std::string& postfix = "",
    const std::string& ext = ""
    );

template <
    class T,
    typename std::enable_if< std::is_floating_point< T >::value >::type * =␣
↪nullptr
    >
void writePreciseString(
    std::ostream& os,
    T value
```

```
    );

template <
    class T,
    typename std::enable_if< std::is_floating_point< T >::value >::type * =␣
↪nullptr
    >
std::string writePreciseString(T value);

template <typename NameIter>
void value_substitutions(
    std::string& code,
    NameIter namesBegin,
    NameIter namesEnd,
    const std::vector<double>& values,
    const std::string& ext = ""
    );

void value_substitutions(
    std::string& code,
    const std::vector<std::string>& names,
    const std::vector<double>& values,
    const std::string& ext = ""
    );

std::string ensureFtype(const std::string& oldcode, const std::string& type);
void checkUnreplacedVariables(const std::string& code, const std::string&␣
↪codeName);

void preNeuronSubstitutionsInSynapticCode(
    std::string& wCode,
    const SynapseGroupInternal& sg,
    const std::string& offset,
    const std::string& axonalDelayOffset,
    const std::string& postIdx,
    const std::string& devPrefix,
    const std::string& preVarPrefix = "",
    const std::string& preVarSuffix = ""
    );

void postNeuronSubstitutionsInSynapticCode(
    std::string& wCode,
    const SynapseGroupInternal& sg,
    const std::string& offset,
    const std::string& backPropDelayOffset,
    const std::string& preIdx,
    const std::string& devPrefix,
    const std::string& postVarPrefix = "",
    const std::string& postVarSuffix = ""
    );

void neuronSubstitutionsInSynapticCode(
    std::string& wCode,
    const SynapseGroupInternal& sg,
```

```
    const std::string& preIdx,
    const std::string& postIdx,
    const std::string& devPrefix,
    double dt,
    const std::string& preVarPrefix = "",
    const std::string& preVarSuffix = "",
    const std::string& postVarPrefix = "",
    const std::string& postVarSuffix = ""
    );

GENN_EXPORT std::ostream& operator << (
    std::ostream& s,
    const CodeStream::OB& ob
    );

GENN_EXPORT std::ostream& operator << (
    std::ostream& s,
    const CodeStream::CB& cb
    );

GENN_EXPORT std::vector<std::string> generateAll(
    const ModelSpecInternal& model,
    const BackendBase& backend,
    const filesystem::path& outputPath,
    bool standaloneModules = false
    );

void generateInit(
    CodeStream& os,
    const ModelSpecInternal& model,
    const BackendBase& backend,
    bool standaloneModules
    );

void GENN_EXPORT generateMakefile(
    std::ostream& os,
    const BackendBase& backend,
    const std::vector<std::string>& moduleNames
    );

void GENN_EXPORT generateMPI(CodeStream& os, const ModelSpecInternal& model,␣
↪const BackendBase& backend, bool standaloneModules);

void GENN_EXPORT generateMSBuild(
    std::ostream& os,
    const BackendBase& backend,
    const std::string& projectGUID,
    const std::vector<std::string>& moduleNames
    );

void generateNeuronUpdate(
    CodeStream& os,
    const ModelSpecInternal& model,
    const BackendBase& backend,
```

```
    bool standaloneModules
    );

MemAlloc generateRunner(
    CodeStream& definitions,
    CodeStream& definitionsInternal,
    CodeStream& runner,
    const ModelSpecInternal& model,
    const BackendBase& backend,
    int localHostID
    );

void generateSupportCode(
    CodeStream& os,
    const ModelSpecInternal& model
    );

void generateSynapseUpdate(
    CodeStream& os,
    const ModelSpecInternal& model,
    const BackendBase& backend,
    bool standaloneModules
    );

 // namespace CodeGenerator
```

28.1.14 Detailed Documentation

Helper class for generating code - automatically inserts brackets, indents etc.

Based heavily on: https://stackoverflow.com/questions/15053753/writing-a-manipulator-for-a-custom-stream-class

Global Functions

```
void substitute(std::string& s, const std::string& trg, const std::string&␣
↪rep)
```

Tool for substituting strings in the neuron code strings or other templates.

```
bool regexVarSubstitute(
    std::string& s,
    const std::string& trg,
    const std::string& rep
    )
```

Tool for substituting variable names in the neuron code strings or other templates using regular expressions.

```
bool regexFuncSubstitute(
    std::string& s,
    const std::string& trg,
    const std::string& rep
    )
```

Tool for substituting function names in the neuron code strings or other templates using regular expressions.

```
void functionSubstitute(
    std::string& code,
    const std::string& funcName,
    unsigned int numParams,
    const std::string& replaceFuncTemplate
    )
```

This function substitutes function calls in the form:

$(functionName, parameter1, param2Function(0.12, “string”))

with replacement templates in the form:

actualFunction(CONSTANT, $(0), $(1))

```
template <typename NameIter>
void name_substitutions(
    std::string& code,
    const std::string& prefix,
    NameIter namesBegin,
    NameIter namesEnd,
    const std::string& postfix = "",
    const std::string& ext = ""
    )
```

This function performs a list of name substitutions for variables in code snippets.

```
void name_substitutions(
    std::string& code,
    const std::string& prefix,
    const std::vector<std::string>& names,
    const std::string& postfix = "",
    const std::string& ext = ""
    )
```

This function performs a list of name substitutions for variables in code snippets.

```
template <
    class T,
    typename std::enable_if< std::is_floating_point< T >::value >::type * =␣
↪nullptr
    >
void writePreciseString(
    std::ostream& os,
    T value
    )
```

This function writes a floating point value to a stream -setting the precision so no digits are lost.

```
template <
    class T,
    typename std::enable_if< std::is_floating_point< T >::value >::type * =␣
↪nullptr
    >
std::string writePreciseString(T value)
```

This function writes a floating point value to a string - setting the precision so no digits are lost.

```
template <typename NameIter>
void value_substitutions(
```

```
    std::string& code,
    NameIter namesBegin,
    NameIter namesEnd,
    const std::vector<double>& values,
    const std::string& ext = ""
    )
```

This function performs a list of value substitutions for parameters in code snippets.

```
void value_substitutions(
    std::string& code,
    const std::vector<std::string>& names,
    const std::vector<double>& values,
    const std::string& ext = ""
    )
```

This function performs a list of value substitutions for parameters in code snippets.

```
std::string ensureFtype(const std::string& oldcode, const std::string& type)
```

This function implements a parser that converts any floating point constant in a code snippet to a floating point constant with an explicit precision (by appending "f" or removing it).

```
void checkUnreplacedVariables(
    const std::string& code,
    const std::string& codeName
    )
```

This function checks for unknown variable definitions and returns a gennError if any are found.

```
void preNeuronSubstitutionsInSynapticCode(
    std::string& wCode,
    const SynapseGroupInternal& sg,
    const std::string& offset,
    const std::string& axonalDelayOffset,
    const std::string& postIdx,
    const std::string& devPrefix,
    const std::string& preVarPrefix = "",
    const std::string& preVarSuffix = ""
    )
```

suffix to be used for presynaptic variable accesses - typically combined with prefix to wrap in function call such as __ldg(&XXX)

Function for performing the code and value substitutions necessary to insert neuron related variables, parameters, and extraGlobal parameters into synaptic code.

```
void postNeuronSubstitutionsInSynapticCode(
    std::string& wCode,
    const SynapseGroupInternal& sg,
    const std::string& offset,
    const std::string& backPropDelayOffset,
    const std::string& preIdx,
    const std::string& devPrefix,
    const std::string& postVarPrefix = "",
    const std::string& postVarSuffix = ""
    )
```

suffix to be used for postsynaptic variable accesses - typically combined with prefix to wrap in function call such as __ldg(&XXX)

```
void neuronSubstitutionsInSynapticCode(
    std::string& wCode,
    const SynapseGroupInternal& sg,
    const std::string& preIdx,
    const std::string& postIdx,
    const std::string& devPrefix,
    double dt,
    const std::string& preVarPrefix = "",
    const std::string& preVarSuffix = "",
    const std::string& postVarPrefix = "",
    const std::string& postVarSuffix = ""
    )
```

Function for performing the code and value substitutions necessary to insert neuron related variables, parameters, and extraGlobal parameters into synaptic code.

suffix to be used for postsynaptic variable accesses - typically combined with prefix to wrap in function call such as __ldg(&XXX)

```
void GENN_EXPORT generateMPI(CodeStream& os, const ModelSpecInternal& model,␣
↪const BackendBase& backend, bool standaloneModules)
```

A function that generates predominantly MPI infrastructure code.

In this function MPI infrastructure code are generated, including: MPI send and receive functions.

28.2 namespace CurrentSourceModels

28.2.1 class CurrentSourceModels::Base

Overview

Base class for all current source models. *More...*

```
#include <currentSourceModels.h>

class Base: public Models::Base

public:
    // methods

    virtual std::string getInjectionCode() const;
;

// direct descendants

class DC;
class GaussianNoise;
```

Inherited Members

```
public:
    // typedefs
```

```
    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
```

Detailed Documentation

Base class for all current source models.

Methods

```
virtual std::string getInjectionCode() const
```

Gets the code that defines current injected each timestep.

28.2.2 class CurrentSourceModels::DC

Overview

DC source. *More...*

```
#include <currentSourceModels.h>

class DC: public CurrentSourceModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<1> ParamValues;
    typedef Models::VarInitContainerBase<0> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods
```

```
    static const DC* getInstance();
    SET_INJECTION_CODE("$(injectCurrent, $(amp));");
    virtual StringVec getParamNames() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getInjectionCode() const;
```

Detailed Documentation

DC source.

It has a single parameter:

- `amp` - amplitude of the current [nA]

Methods

```
virtual StringVec getParamNames() const
```

Gets names of of (independent) model parameters.

28.2.3 class CurrentSourceModels::GaussianNoise

Overview

Noisy current source with noise drawn from normal distribution. *More...*

```
#include <currentSourceModels.h>

class GaussianNoise: public CurrentSourceModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<2> ParamValues;
    typedef Models::VarInitContainerBase<0> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const GaussianNoise* getInstance();
    SET_INJECTION_CODE("$(injectCurrent, $(mean) + $(gennrand_normal)* $(sd));
↪");
    virtual StringVec getParamNames() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getInjectionCode() const;
```

Detailed Documentation

Noisy current source with noise drawn from normal distribution.

It has 2 parameters:

- `mean` - mean of the normal distribution [nA]
- `sd` - standard deviation of the normal distribution [nA]

Methods

```
virtual StringVec getParamNames() const
```

Gets names of of (independent) model parameters.

```
namespace CurrentSourceModels

// classes

class Base;
class DC;
class GaussianNoise;

 // namespace CurrentSourceModels
```

28.3 namespace InitSparseConnectivitySnippet

28.3.1 class InitSparseConnectivitySnippet::Base

Overview

```
#include <initSparseConnectivitySnippet.h>

class Base: public Snippet::Base

public:
    // typedefs

    typedef std::function<unsigned int(unsigned int, unsigned int, const␣
↪std::vector<double>&)> CalcMaxLengthFunc;

    // methods

    virtual std::string getRowBuildCode() const;
    virtual ParamValVec getRowBuildStateVars() const;
    virtual CalcMaxLengthFunc getCalcMaxRowLengthFunc() const;
    virtual CalcMaxLengthFunc getCalcMaxColLengthFunc() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
;

// direct descendants
```

```
class FixedProbabilityBase;
class OneToOne;
class Uninitialised;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
```

Detailed Documentation

Methods

`virtual CalcMaxLengthFunc getCalcMaxRowLengthFunc() const`

Get function to calculate the maximum row length of this connector based on the parameters and the size of the pre and postsynaptic population.

`virtual CalcMaxLengthFunc getCalcMaxColLengthFunc() const`

Get function to calculate the maximum column length of this connector based on the parameters and the size of the pre and postsynaptic population.

`virtual EGPVec getExtraGlobalParams() const`

Gets names and types (as strings) of additional per-population parameters for the connection initialisation snippet

`size_t getExtraGlobalParamIndex(const std::string& paramName) const`

Find the index of a named extra global parameter.

28.3.2 class InitSparseConnectivitySnippet::FixedProbability

Initialises connectivity with a fixed probability of a synapse existing between a pair of pre and postsynaptic neurons.

Whether a synapse exists between a pair of pre and a postsynaptic neurons can be modelled using a Bernoulli distribution. While this COULD br sampling directly by repeatedly drawing from the uniform distribution, this is inneffi-cient. Instead we sample from the gemetric distribution which describes "the probability distribution of the number of Bernoulli trials needed to get one success" essentially the distribution of the 'gaps' between synapses. We do this using the "inversion method" described by Devroye (1986) essentially inverting the CDF of the equivalent continuous distribution (in this case the exponential distribution)

```
#include <initSparseConnectivitySnippet.h>

class FixedProbability: public InitSparseConnectivitySnippet::FixedProbabilityBase

public:
    // methods

    DECLARE_SNIPPET(
        InitSparseConnectivitySnippet::FixedProbability,
        1
        );

    SET_ROW_BUILD_CODE();
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::function<unsigned int(unsigned int, unsigned int, const␣
↪std::vector<double>&)> CalcMaxLengthFunc;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getRowBuildCode() const;
    virtual ParamValVec getRowBuildStateVars() const;
    virtual CalcMaxLengthFunc getCalcMaxRowLengthFunc() const;
    virtual CalcMaxLengthFunc getCalcMaxColLengthFunc() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getRowBuildCode() const = 0;
    SET_ROW_BUILD_STATE_VARS("prevJ","int",-1);
    virtual StringVec getParamNames() const;
```

```
    virtual DerivedParamVec getDerivedParams() const;
    SET_CALC_MAX_ROW_LENGTH_FUNC((unsigned int numPre, unsigned int numPost,
↪ const std::vector<double>&pars)const double quantile=pow(0.9999, 1.0/
↪(double) numPre);return binomialInverseCDF(quantile, numPost, pars[0]););
    SET_CALC_MAX_COL_LENGTH_FUNC((unsigned int numPre, unsigned int numPost,
↪ const std::vector<double>&pars)const double quantile=pow(0.9999, 1.0/
↪(double) numPost);return binomialInverseCDF(quantile, numPre, pars[0]););
```

28.3.3 class InitSparseConnectivitySnippet::FixedProbabilityBase

Overview

Base class for snippets which initialise connectivity with a fixed probability of a synapse existing between a pair of pre and postsynaptic neurons. *More...*

```
#include <initSparseConnectivitySnippet.h>

class FixedProbabilityBase: public InitSparseConnectivitySnippet::Base

public:
    // methods

    virtual std::string getRowBuildCode() const = 0;
    SET_ROW_BUILD_STATE_VARS("prevJ","int",-1);
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    SET_CALC_MAX_ROW_LENGTH_FUNC((unsigned int numPre, unsigned int numPost,
↪ const std::vector<double>&pars)const double quantile=pow(0.9999, 1.0/
↪(double) numPre);return binomialInverseCDF(quantile, numPost, pars[0]););
    SET_CALC_MAX_COL_LENGTH_FUNC((unsigned int numPre, unsigned int numPost,
↪ const std::vector<double>&pars)const double quantile=pow(0.9999, 1.0/
↪(double) numPost);return binomialInverseCDF(quantile, numPre, pars[0]););
;

// direct descendants

class FixedProbability;
class FixedProbabilityNoAutapse;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::function<unsigned int(unsigned int, unsigned int, const␣
↪std::vector<double>&)> CalcMaxLengthFunc;

    // structs
```

```
    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getRowBuildCode() const;
    virtual ParamValVec getRowBuildStateVars() const;
    virtual CalcMaxLengthFunc getCalcMaxRowLengthFunc() const;
    virtual CalcMaxLengthFunc getCalcMaxColLengthFunc() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
```

Detailed Documentation

Base class for snippets which initialise connectivity with a fixed probability of a synapse existing between a pair of pre and postsynaptic neurons.

Methods

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

`virtual DerivedParamVec getDerivedParams() const`

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

28.3.4 class InitSparseConnectivitySnippet::FixedProbabilityNoAutapse

Initialises connectivity with a fixed probability of a synapse existing between a pair of pre and postsynaptic neurons. This version ensures there are no autapses - connections between neurons with the same id so should be used for recurrent connections.

Whether a synapse exists between a pair of pre and a postsynaptic neurons can be modelled using a Bernoulli distribution. While this COULD br sampling directly by repeatedly drawing from the uniform distribution, this is inneffi-cient. Instead we sample from the gemetric distribution which describes "the probability distribution of the number of Bernoulli trials needed to get one success" essentially the distribution of the 'gaps' between synapses. We do this using the "inversion method" described by Devroye (1986) essentially inverting the CDF of the equivalent continuous distribution (in this case the exponential distribution)

```
#include <initSparseConnectivitySnippet.h>

class FixedProbabilityNoAutapse: public InitSparseConnectivitySnippet::FixedProbabilityBase

public:
    // methods

    DECLARE_SNIPPET(
```

```
        InitSparseConnectivitySnippet::FixedProbabilityNoAutapse,
        1
        );

    SET_ROW_BUILD_CODE();
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::function<unsigned int(unsigned int, unsigned int, const␣
↪std::vector<double>&)> CalcMaxLengthFunc;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getRowBuildCode() const;
    virtual ParamValVec getRowBuildStateVars() const;
    virtual CalcMaxLengthFunc getCalcMaxRowLengthFunc() const;
    virtual CalcMaxLengthFunc getCalcMaxColLengthFunc() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getRowBuildCode() const = 0;
    SET_ROW_BUILD_STATE_VARS("prevJ","int",-1);
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    SET_CALC_MAX_ROW_LENGTH_FUNC((unsigned int numPre, unsigned int numPost,
↪ const std::vector<double>&pars)const double quantile=pow(0.9999, 1.0/
↪(double) numPre);return binomialInverseCDF(quantile, numPost, pars[0]););
    SET_CALC_MAX_COL_LENGTH_FUNC((unsigned int numPre, unsigned int numPost,
↪ const std::vector<double>&pars)const double quantile=pow(0.9999, 1.0/
↪(double) numPost);return binomialInverseCDF(quantile, numPre, pars[0]););
```

28.3.5 class InitSparseConnectivitySnippet::Init

```
#include <initSparseConnectivitySnippet.h>

class Init: public Snippet::Init
```

```
public:
    // methods

    Init(
        const Base* snippet,
        const std::vector<double>& params
        );
;
```

Inherited Members

```
public:
    // methods

    Init(const SnippetBase* snippet, const std::vector<double>& params);
    const SnippetBase* getSnippet() const;
    const std::vector<double>& getParams() const;
    const std::vector<double>& getDerivedParams() const;
    void initDerivedParams(double dt);
```

28.3.6 class InitSparseConnectivitySnippet::OneToOne

Initialises connectivity to a 'one-to-one' diagonal matrix.

```
#include <initSparseConnectivitySnippet.h>

class OneToOne: public InitSparseConnectivitySnippet::Base

public:
    // methods

    DECLARE_SNIPPET(
        InitSparseConnectivitySnippet::OneToOne,
        0
        );

    SET_ROW_BUILD_CODE("$(addSynapse, $(id_pre));""$(endRow);");
    SET_MAX_ROW_LENGTH(1);
    SET_MAX_COL_LENGTH(1);
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
```

```
    typedef std::function<unsigned int(unsigned int, unsigned int, const␣
↪std::vector<double>&)> CalcMaxLengthFunc;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getRowBuildCode() const;
    virtual ParamValVec getRowBuildStateVars() const;
    virtual CalcMaxLengthFunc getCalcMaxRowLengthFunc() const;
    virtual CalcMaxLengthFunc getCalcMaxColLengthFunc() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
```

28.3.7 class InitSparseConnectivitySnippet::Uninitialised

Used to mark connectivity as uninitialised - no initialisation code will be run.

```
#include <initSparseConnectivitySnippet.h>

class Uninitialised: public InitSparseConnectivitySnippet::Base

public:
    // methods

    DECLARE_SNIPPET(
        InitSparseConnectivitySnippet::Uninitialised,
        0
        );
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::function<unsigned int(unsigned int, unsigned int, const␣
↪std::vector<double>&)> CalcMaxLengthFunc;

    // structs

    struct DerivedParam;
```

```
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getRowBuildCode() const;
    virtual ParamValVec getRowBuildStateVars() const;
    virtual CalcMaxLengthFunc getCalcMaxRowLengthFunc() const;
    virtual CalcMaxLengthFunc getCalcMaxColLengthFunc() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
```

Base class for all sparse connectivity initialisation snippets.

```
namespace InitSparseConnectivitySnippet

// classes

class Base;
class FixedProbability;
class FixedProbabilityBase;
class FixedProbabilityNoAutapse;
class Init;
class OneToOne;
class Uninitialised;

 // namespace InitSparseConnectivitySnippet
```

28.4 namespace InitVarSnippet

28.4.1 class InitVarSnippet::Base

```
#include <initVarSnippet.h>

class Base: public Snippet::Base

public:
    // methods

    virtual std::string getCode() const;
;

// direct descendants

class Constant;
class Exponential;
class Gamma;
class Normal;
class Uniform;
class Uninitialised;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
```

28.4.2 class InitVarSnippet::Constant

Overview

Initialises variable to a constant value. *More...*

```
#include <initVarSnippet.h>

class Constant: public InitVarSnippet::Base

public:
    // methods

    DECLARE_SNIPPET(
        InitVarSnippet::Constant,
        1
        );

    SET_CODE();
    virtual StringVec getParamNames() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
```

```
    typedef std::vector<DerivedParam> DerivedParamVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getCode() const;
```

Detailed Documentation

Initialises variable to a constant value.

This snippet takes 1 parameter:

- `value` - The value to intialise the variable to

This snippet type is seldom used directly - *Models::VarInit* has an implicit constructor that, internally, creates one of these snippets

Methods

```
virtual StringVec getParamNames() const
```

Gets names of of (independent) model parameters.

28.4.3 class InitVarSnippet::Exponential

Overview

Initialises variable by sampling from the exponential distribution. *More...*

```
#include <initVarSnippet.h>

class Exponential: public InitVarSnippet::Base

public:
    // methods

    DECLARE_SNIPPET(
        InitVarSnippet::Exponential,
        1
        );

    SET_CODE();
    virtual StringVec getParamNames() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getCode() const;
```

Detailed Documentation

Initialises variable by sampling from the exponential distribution.

This snippet takes 1 parameter:

- `lambda` - mean event rate (events per unit time/distance)

Methods

```
virtual StringVec getParamNames() const
```

Gets names of of (independent) model parameters.

28.4.4 class InitVarSnippet::Gamma

Overview

Initialises variable by sampling from the exponential distribution. *More...*

```
#include <initVarSnippet.h>

class Gamma: public InitVarSnippet::Base

public:
    // methods

    DECLARE_SNIPPET(
        InitVarSnippet::Gamma,
        2
```

```
        );

    SET_CODE();
    virtual StringVec getParamNames() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getCode() const;
```

Detailed Documentation

Initialises variable by sampling from the exponential distribution.

This snippet takes 1 parameter:

- `lambda` - mean event rate (events per unit time/distance)

Methods

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

28.4.5 class InitVarSnippet::Normal

Overview

Initialises variable by sampling from the normal distribution. *More...*

```
#include <initVarSnippet.h>

class Normal: public InitVarSnippet::Base
```

```
public:
    // methods

    DECLARE_SNIPPET(
        InitVarSnippet::Normal,
        2
        );

    SET_CODE();
    virtual StringVec getParamNames() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getCode() const;
```

Detailed Documentation

Initialises variable by sampling from the normal distribution.

This snippet takes 2 parameters:

- `mean` - The mean
- `sd` - The standard distribution

Methods

```
virtual StringVec getParamNames() const
```

Gets names of of (independent) model parameters.

28.4.6 class InitVarSnippet::Uniform

Overview

Initialises variable by sampling from the uniform distribution. *More...*

```
#include <initVarSnippet.h>

class Uniform: public InitVarSnippet::Base

public:
    // methods

    DECLARE_SNIPPET(
        InitVarSnippet::Uniform,
        2
        );

    SET_CODE();
    virtual StringVec getParamNames() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getCode() const;
```

Detailed Documentation

Initialises variable by sampling from the uniform distribution.

This snippet takes 2 parameters:

- `min` - The minimum value
- `max` - The maximum value

Methods

```
virtual StringVec getParamNames() const
```

Gets names of of (independent) model parameters.

28.4.7 class InitVarSnippet::Uninitialised

Used to mark variables as uninitialised - no initialisation code will be run.

```
#include <initVarSnippet.h>

class Uninitialised: public InitVarSnippet::Base

public:
    // methods

    DECLARE_SNIPPET(
        InitVarSnippet::Uninitialised,
        0
        );
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual std::string getCode() const;
```

Base class for all value initialisation snippets.

```
namespace InitVarSnippet

// classes

class Base;
class Constant;
```

```
class Exponential;
class Gamma;
class Normal;
class Uniform;
class Uninitialised;

 // namespace InitVarSnippet
```

28.5 namespace Models

28.5.1 class Models::Base

struct Models::Base::Var

Overview

A variable has a name, a type and an access type. *More...*

```
#include <models.h>

struct Var

    // fields

    std::string name;
    std::string type;
    VarAccess access;

    // methods

    Var();

    Var(
        const std::string& n,
        const std::string& t,
        VarAccess a
        );

    Var(
        const std::string& n,
        const std::string& t
        );
;
```

Detailed Documentation

A variable has a name, a type and an access type.

Explicit constructors required as although, through the wonders of C++ aggregate initialization, access would default to *VarAccess::READ_WRITE* if not specified, this results in a -Wmissing-field-initializers warning on GCC and Clang

Overview

Base class for all models - in addition to the parameters snippets have, models can have state variables. *More...*

```
#include <models.h>

class Base: public Snippet::Base

public:
    // typedefs

    typedef std::vector<Var> VarVec;

    // structs

    struct Var;

    // methods

    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
;

// direct descendants

class Base;
class Base;
class Base;
class Base;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
```

Detailed Documentation

Base class for all models - in addition to the parameters snippets have, models can have state variables.

Methods

`virtual VarVec getVars() const`

Gets names and types (as strings) of model variables.

`virtual EGPVec getExtraGlobalParams() const`

Gets names and types (as strings) of additional per-population parameters for the weight update model.

`size_t getVarIndex(const std::string& varName) const`

Find the index of a named variable.

`size_t getExtraGlobalParamIndex(const std::string& paramName) const`

Find the index of a named extra global parameter.

28.5.2 class Models::VarInit

```
#include <models.h>

class VarInit: public Snippet::Init

public:
    // methods

    VarInit(
        const InitVarSnippet::Base* snippet,
        const std::vector<double>& params
        );

    VarInit(double constant);
;
```

Inherited Members

```
public:
    // methods

    Init(const SnippetBase* snippet, const std::vector<double>& params);
    const SnippetBase* getSnippet() const;
    const std::vector<double>& getParams() const;
    const std::vector<double>& getDerivedParams() const;
    void initDerivedParams(double dt);
```

28.5.3 template class Models::VarInitContainerBase<0>

Overview

Template specialisation of ValueInitBase to avoid compiler warnings in the case when a model requires no variable initialisers *More...*

```
#include <models.h>

template <>
class VarInitContainerBase<0>

public:
    // methods

    template <typename... T>
    VarInitContainerBase(T&&... initialisers);

    VarInitContainerBase(const Snippet::ValueBase<0>&);
    std::vector<VarInit> getInitialisers() const;
;
```

Detailed Documentation

Template specialisation of ValueInitBase to avoid compiler warnings in the case when a model requires no variable initialisers

Methods

```
std::vector<VarInit> getInitialisers() const
```

Gets initialisers as a vector of Values.

28.5.4 template class Models::VarInitContainerBase

Overview

Wrapper to ensure at compile time that correct number of value initialisers are used when specifying the values of a model's initial state. *More...*

```
#include <models.h>

template <size_t NumVars>
class VarInitContainerBase

public:
    // methods

    template <typename... T>
    VarInitContainerBase(T&&... initialisers);

    const std::vector<VarInit>& getInitialisers() const;
    const VarInit& operator [] (size_t pos) const;
;
```

Detailed Documentation

Wrapper to ensure at compile time that correct number of value initialisers are used when specifying the values of a model's initial state.

Methods

```
const std::vector<VarInit>& getInitialisers() const
```

Gets initialisers as a vector of Values.

Class used to bind together everything required to initialise a variable:

1. A pointer to a variable initialisation snippet
2. The parameters required to control the variable initialisation snippet

```
namespace Models

// classes

class Base;
class VarInit;

template <>
class VarInitContainerBase<0>;

template <size_t NumVars>
class VarInitContainerBase;

 // namespace Models
```

28.6 namespace NeuronModels

28.6.1 class NeuronModels::Base

Overview

Base class for all neuron models. *More...*

```
#include <neuronModels.h>

class Base: public Models::Base

public:
    // methods

    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
;
```

```
// direct descendants

class Izhikevich;
class LIF;
class Poisson;
class PoissonNew;
class RulkovMap;
class SpikeSource;
class SpikeSourceArray;
class TraubMiles;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
```

Detailed Documentation

Base class for all neuron models.

Methods

```
virtual std::string getSimCode() const
```

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name "NN". It needs to refer to the predefined variable "ISYN", i.e. contain , if it is to receive input.

`virtual std::string getThresholdConditionCode() const`

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

`virtual std::string getResetCode() const`

Gets code that defines the reset action taken after a spike occurred. This can be empty.

`virtual std::string getSupportCode() const`

Gets support code to be made available within the neuron kernel/funcion.

This is intended to contain user defined device functions that are used in the neuron codes. Preprocessor defines are also allowed if appropriately safeguarded against multiple definition by using ifndef; functions should be declared as "__host__ __device__" to be available for both GPU and CPU versions.

`virtual Models::Base::ParamValVec getAdditionalInputVars() const`

Gets names, types (as strings) and initial values of local variables into which the 'apply input code' of (potentially) multiple postsynaptic input models can apply input

`virtual bool isAutoRefractoryRequired() const`

Does this model require auto-refractory logic?

28.6.2 class NeuronModels::Izhikevich

Overview

Izhikevich neuron with fixed parameters izhikevich2003simple. *More...*

```
#include <neuronModels.h>

class Izhikevich: public NeuronModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<4> ParamValues;
    typedef Models::VarInitContainerBase<2> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::Izhikevich* getInstance();
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual StringVec getParamNames() const;
    virtual VarVec getVars() const;
;

// direct descendants

class IzhikevichVariable;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
```

Detailed Documentation

Izhikevich neuron with fixed parameters izhikevich2003simple.

It is usually described as

$$to$$

$$\frac{dV}{dt} = 0.04V^2 + 5V + 140 - U + I,$$

$$\frac{dU}{dt} = a(bV - U),$$

$$=$$

$0.04V^2 + 5V + 140 - U + I$, $\frac{dU}{dt} =$
$a(bV - U)$,

I is an external input current and the voltage V is reset to parameter c and U incremented by parameter d, whenever V >= 30 mV. This is paired with a particular integration procedure of two 0.5 ms Euler time steps for the V equation followed by one 1 ms time step of the U equation. Because of its popularity we provide this model in this form here event though due to the details of the usual implementation it is strictly speaking inconsistent with the displayed equations.

Variables are:

- `V` - Membrane potential
- `U` - Membrane recovery variable

Parameters are:

- `a` - time scale of U
- `b` - sensitivity of U
- `c` - after-spike reset value of V
- `d` - after-spike reset value of U

Methods

`virtual std::string getSimCode() const`

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name "NN". It needs to refer to the predefined variable "ISYN", i.e. contain , if it is to receive input.

`virtual std::string getThresholdConditionCode() const`

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

`virtual VarVec getVars() const`

Gets names and types (as strings) of model variables.

28.6.3 class NeuronModels::IzhikevichVariable

Overview

Izhikevich neuron with variable parameters izhikevich2003simple. *More...*

```
#include <neuronModels.h>

class IzhikevichVariable: public NeuronModels::Izhikevich

public:
    // typedefs

    typedef Snippet::ValueBase<0> ParamValues;
    typedef Models::VarInitContainerBase<6> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::IzhikevichVariable* getInstance();
    virtual StringVec getParamNames() const;
    virtual VarVec getVars() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;
    typedef Snippet::ValueBase<4> ParamValues;
    typedef Models::VarInitContainerBase<2> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
```

```
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
    static const NeuronModels::Izhikevich* getInstance();
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual StringVec getParamNames() const;
    virtual VarVec getVars() const;
```

Detailed Documentation

Izhikevich neuron with variable parameters izhikevich2003simple.

This is the same model as *Izhikevich* but parameters are defined as "variables" in order to allow users to provide individual values for each individual neuron instead of fixed values for all neurons across the population.

Accordingly, the model has the Variables:

- `V` - Membrane potential
- `U` - Membrane recovery variable
- `a` - time scale of U
- `b` - sensitivity of U
- `c` - after-spike reset value of V
- `d` - after-spike reset value of U

and no parameters.

Methods

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

`virtual VarVec getVars() const`

Gets names and types (as strings) of model variables.

28.6.4 class NeuronModels::LIF

Overview

```
#include <neuronModels.h>

class LIF: public NeuronModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<7> ParamValues;
    typedef Models::VarInitContainerBase<2> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;
```

```
    // methods

    static const LIF* getInstance();
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    SET_NEEDS_AUTO_REFRACTORY(false);
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
```

Detailed Documentation

Methods

```
virtual std::string getSimCode() const
```

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name "NN". It needs to refer to the predefined variable "ISYN", i.e. contain , if it is to receive input.

```
virtual std::string getThresholdConditionCode() const
```

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

```
virtual std::string getResetCode() const
```

Gets code that defines the reset action taken after a spike occurred. This can be empty.

```
virtual StringVec getParamNames() const
```

Gets names of of (independent) model parameters.

```
virtual DerivedParamVec getDerivedParams() const
```

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

```
virtual VarVec getVars() const
```

Gets names and types (as strings) of model variables.

28.6.5 class NeuronModels::Poisson

Overview

Poisson neurons. *More...*

```
#include <neuronModels.h>

class Poisson: public NeuronModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<4> ParamValues;
    typedef Models::VarInitContainerBase<2> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::Poisson* getInstance();
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual StringVec getParamNames() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
```

Detailed Documentation

Poisson neurons.

Poisson neurons have constant membrane potential (`Vrest`) unless they are activated randomly to the `Vspike` value if (t- `SpikeTime`) > `trefract`.

It has 2 variables:

- `V` - Membrane potential (mV)
- `SpikeTime` - Time at which the neuron spiked for the last time (ms)

and 4 parameters:

- `trefract` - Refractory period (ms)
- `tspike` - duration of spike (ms)
- `Vspike` - Membrane potential at spike (mV)
- `Vrest` - Membrane potential at rest (mV)

The initial values array for the `Poisson` type needs two entries for `V`, and `SpikeTime` and the parameter array needs four entries for `therate`, `trefract`, `Vspike` and `Vrest`, *in that order*.

This model uses a linear approximation for the probability of firing a spike in a given time step of size `DT`, i.e. the probability of firing is λ times `DT` : $p = \lambda \Delta t$. This approximation is usually very good, especially for typical, quite small time steps and moderate firing rates. However, it is worth noting that the approximation becomes poor for very high firing rates and large time steps.

Methods

`virtual std::string getSimCode() const`

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name "NN". It needs to refer to the predefined variable "ISYN", i.e. contain , if it is to receive input.

`virtual std::string getThresholdConditionCode() const`

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

`virtual VarVec getVars() const`

Gets names and types (as strings) of model variables.

`virtual EGPVec getExtraGlobalParams() const`

Gets names and types (as strings) of additional per-population parameters for the weight update model.

28.6.6 class NeuronModels::PoissonNew

Overview

Poisson neurons. *More...*

```
#include <neuronModels.h>

class PoissonNew: public NeuronModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<1> ParamValues;
    typedef Models::VarInitContainerBase<1> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::PoissonNew* getInstance();
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
```

```
    virtual StringVec getParamNames() const;
    virtual VarVec getVars() const;
    virtual DerivedParamVec getDerivedParams() const;
    SET_NEEDS_AUTO_REFRACTORY(false);
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
```

Detailed Documentation

Poisson neurons.

It has 1 state variable:

- `timeStepToSpike` - Number of timesteps to next spike

and 1 parameter:

- `rate` - Mean firing rate (Hz)

Internally this samples from the exponential distribution using the C++ 11 <random> library on the CPU and by transforming the uniform distribution, generated using cuRAND, with a natural log on the GPU.

Methods

`virtual std::string getSimCode() const`

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name "NN". It needs to refer to the predefined variable "ISYN", i.e. contain , if it is to receive input.

`virtual std::string getThresholdConditionCode() const`

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

`virtual VarVec getVars() const`

Gets names and types (as strings) of model variables.

`virtual DerivedParamVec getDerivedParams() const`

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

28.6.7 class NeuronModels::RulkovMap

Overview

Rulkov Map neuron. *More...*

```
#include <neuronModels.h>

class RulkovMap: public NeuronModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<4> ParamValues;
    typedef Models::VarInitContainerBase<2> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::RulkovMap* getInstance();
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual StringVec getParamNames() const;
    virtual VarVec getVars() const;
    virtual DerivedParamVec getDerivedParams() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
```

Detailed Documentation

Rulkov Map neuron.

The *RulkovMap* type is a map based neuron model based on Rulkov2002 but in the 1-dimensional map form used in nowotny2005self :

$$to$$

$$V(t+\Delta t) =$$

$$\begin{cases} V_{\text{spike}}\left(\frac{\alpha V_{\text{spike}}}{V_{\text{spike}}-V(t)\beta I_{\text{syn}}}+y\right) & V(t)\leq 0 \\ V_{\text{spike}}\left(\alpha+y\right) & V(t)\leq V_{\text{spike}}\left(\alpha+y\right)\ \&\ V(t-\Delta t)\leq 0 \\ -V_{\text{spike}} & \text{otherwise} \end{cases}$$

$$=$$

$$\begin{cases} V_{\text{spike}}\left(\frac{\alpha V_{\text{spike}}}{V_{\text{spike}}-V(t)\beta I_{\text{syn}}}+y\right) & V(t)\leq 0 \\ V_{\text{spike}}\left(\alpha+y\right) & V(t)\leq V_{\text{spike}}\left(\alpha+y\right)\ \&\ V(t-\Delta t)\leq 0 \\ -V_{\text{spike}} & \text{otherwise} \end{cases}$$

The `RulkovMap` type only works as intended for the single time step size of `DT` = 0.5.

The `RulkovMap` type has 2 variables:

- `V` - the membrane potential
- `preV` - the membrane potential at the previous time step

and it has 4 parameters:

- `Vspike` - determines the amplitude of spikes, typically -60mV
- `alpha` - determines the shape of the iteration function, typically α = 3
- `y` - "shift / excitation" parameter, also determines the iteration function,originally, y= -2.468
- `beta` - roughly speaking equivalent to the input resistance, i.e. it regulates the scale of the input into the neuron, typically β = 2.64 MΩ.

The initial values array for the `RulkovMap` type needs two entries for `V` and `Vpre` and the parameter array needs four entries for `Vspike`, `alpha`, `y` and `beta`, *in that order*.

Methods

`virtual std::string getSimCode() const`

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name "NN". It needs to refer to the predefined variable "ISYN", i.e. contain , if it is to receive input.

`virtual std::string getThresholdConditionCode() const`

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

`virtual VarVec getVars() const`

Gets names and types (as strings) of model variables.

`virtual DerivedParamVec getDerivedParams() const`

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

28.6.8 class NeuronModels::SpikeSource

Overview

Empty neuron which allows setting spikes from external sources. *More...*

```
#include <neuronModels.h>

class SpikeSource: public NeuronModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<0> ParamValues;
    typedef Models::VarInitContainerBase<0> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::SpikeSource* getInstance();
    virtual std::string getThresholdConditionCode() const;
    SET_NEEDS_AUTO_REFRACTORY(false);
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
```

```
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
```

Detailed Documentation

Empty neuron which allows setting spikes from external sources.

This model does not contain any update code and can be used to implement the equivalent of a SpikeGeneratorGroup in Brian or a *SpikeSourceArray* in PyNN.

Methods

```
virtual std::string getThresholdConditionCode() const
```

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

28.6.9 class NeuronModels::SpikeSourceArray

Overview

Spike source array. *More...*

```
#include <neuronModels.h>

class SpikeSourceArray: public NeuronModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<0> ParamValues;
    typedef Models::VarInitContainerBase<2> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::SpikeSourceArray* getInstance();
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    SET_NEEDS_AUTO_REFRACTORY(false);
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
```

Detailed Documentation

Spike source array.

A neuron which reads spike times from a global spikes array It has 2 variables:

- `startSpike` - Index of the next spike in the global array
- `endSpike` - Index of the spike next to the last in the globel array

and 1 global parameter:

- `spikeTimes` - Array with all spike times

Methods

```
virtual std::string getSimCode() const
```

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name “NN”. It needs to refer to the predefined variable “ISYN”, i.e. contain , if it is to receive input.

```
virtual std::string getThresholdConditionCode() const
```

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. “V > 20”).

```
virtual std::string getResetCode() const
```

Gets code that defines the reset action taken after a spike occurred. This can be empty.

```
virtual VarVec getVars() const
```

Gets names and types (as strings) of model variables.

```
virtual EGPVec getExtraGlobalParams() const
```

Gets names and types (as strings) of additional per-population parameters for the weight update model.

28.6.10 class NeuronModels::TraubMiles

Overview

Hodgkin-Huxley neurons with Traub & Miles algorithm. *More...*

```
#include <neuronModels.h>

class TraubMiles: public NeuronModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<7> ParamValues;
    typedef Models::VarInitContainerBase<4> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::TraubMiles* getInstance();
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual StringVec getParamNames() const;
    virtual VarVec getVars() const;
;

// direct descendants

class TraubMilesAlt;
class TraubMilesFast;
class TraubMilesNStep;
```

Inherited Members

```
public:
```

```
// typedefs

typedef std::vector<std::string> StringVec;
typedef std::vector<EGP> EGPVec;
typedef std::vector<ParamVal> ParamValVec;
typedef std::vector<DerivedParam> DerivedParamVec;
typedef std::vector<Var> VarVec;

// structs

struct DerivedParam;
struct EGP;
struct ParamVal;
struct Var;

// methods

virtual ~Base();
virtual StringVec getParamNames() const;
virtual DerivedParamVec getDerivedParams() const;
virtual VarVec getVars() const;
virtual EGPVec getExtraGlobalParams() const;
size_t getVarIndex(const std::string& varName) const;
size_t getExtraGlobalParamIndex(const std::string& paramName) const;
virtual std::string getSimCode() const;
virtual std::string getThresholdConditionCode() const;
virtual std::string getResetCode() const;
virtual std::string getSupportCode() const;
virtual Models::Base::ParamValVec getAdditionalInputVars() const;
virtual bool isAutoRefractoryRequired() const;
```

Detailed Documentation

Hodgkin-Huxley neurons with Traub & Miles algorithm.

This conductance based model has been taken from Traub1991 and can be described by the equations:

$$to$$

$$C\frac{dV}{dt} = -I_{\mathrm{Na}} - I_K - I_{\mathrm{leak}} - I_M - I_{i,DC} - I_{i,\mathrm{syn}} - I_i,$$

$$I_{\mathrm{Na}}(t) = g_{\mathrm{Na}} m_i(t)^3 h_i(t)(V_i(t) - E_{\mathrm{Na}})$$

$$I_{\mathrm{K}}(t) = g_{\mathrm{K}} n_i(t)^4 (V_i(t) - E_{\mathrm{K}})$$

$$\frac{dy(t)}{dt} = \alpha_y(V(t))(1 - y(t)) - \beta_y(V(t))y(t),$$

$$
\begin{aligned}
&= \\
-I_{\mathrm{Na}} - I_K - I_{\mathrm{leak}} - I_M - I_{i,DC} - I_{i,\mathrm{syn}} - I_i, I_{\mathrm{Na}}(t) &= \\
g_{\mathrm{Na}} m_i(t)^3 h_i(t)(V_i(t) - E_{\mathrm{Na}}) I_{\mathrm{K}}(t) &= \\
g_{\mathrm{K}} n_i(t)^4 (V_i(t) - E_{\mathrm{K}}) \frac{dy(t)}{dt} &= \\
\alpha_y(V(t))(1 - y(t)) - \beta_y(V(t)) y(t),
\end{aligned}
$$

where $y_i = m, h, n$, and

$$
\begin{aligned}
to & \\
\alpha_n &= \\
0.032(-50 - V)/\big(\exp((-50 - V)/5) - 1\big) & \\
\beta_n &= \\
0.5 \exp((-55 - V)/40) & \\
\alpha_m &= \\
0.32(-52 - V)/\big(\exp((-52 - V)/4) - 1\big) & \\
\beta_m &= \\
0.28(25 + V)/\big(\exp((25 + V)/5) - 1\big) & \\
\alpha_h &= \\
0.128 \exp((-48 - V)/18) & \\
\beta_h &= \\
4/\big(\exp((-25 - V)/5) + 1\big). &
\end{aligned}
$$

$$
\begin{aligned}
&= \\
0.032(-50 - V)/\big(\exp((-50 - V)/5) - 1\big) \beta_n &= \\
0.5 \exp((-55 - V)/40) \alpha_m &= \\
0.32(-52 - V)/\big(\exp((-52 - V)/4) - 1\big) \beta_m &= \\
0.28(25 + V)/\big(\exp((25 + V)/5) - 1\big) \alpha_h &= \\
0.128 \exp((-48 - V)/18) \beta_h &= \\
4/\big(\exp((-25 - V)/5) + 1\big). &
\end{aligned}
$$

and typical parameters are $C = 0.143$ nF, $g_{\mathrm{leak}} = 0.02672$ μ S, $E_{\mathrm{leak}} = -63.563$ mV, $g_{\mathrm{Na}} = 7.15$ μ S, $E_{\mathrm{Na}} = 50$ mV, $g_{\mathrm{K}} = 1.43$ μ S, $E_{\mathrm{K}} = -95$ mV.

It has 4 variables:

- `V` - membrane potential E
- `m` - probability for Na channel activation m
- `h` - probability for not Na channel blocking h
- `n` - probability for K channel activation n

and 7 parameters:

- `gNa` - Na conductance in 1/(mOhms * cm^2)
- `ENa` - Na equi potential in mV
- `gK` - K conductance in 1/(mOhms * cm^2)
- `EK` - K equi potential in mV
- `gl` - Leak conductance in 1/(mOhms * cm^2)
- `El` - Leak equi potential in mV
- `Cmem` - Membrane capacity density in muF/cm^2

Internally, the ordinary differential equations defining the model are integrated with a linear Euler algorithm and GeNN integrates 25 internal time steps for each neuron for each network time step. I.e., if the network is simulated at `DT= 0.1` ms, then the neurons are integrated with a linear Euler algorithm with `lDT= 0.004` ms. This variant uses IF statements to check for a value at which a singularity would be hit. If so, value calculated by L'Hospital rule is used.

Methods

```
virtual std::string getSimCode() const
```

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name "NN". It needs to refer to the predefined variable "ISYN", i.e. contain , if it is to receive input.

```
virtual std::string getThresholdConditionCode() const
```

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

```
virtual StringVec getParamNames() const
```

Gets names of of (independent) model parameters.

```
virtual VarVec getVars() const
```

Gets names and types (as strings) of model variables.

28.6.11 class NeuronModels::TraubMilesAlt

Overview

Hodgkin-Huxley neurons with Traub & Miles algorithm. *More...*

```
#include <neuronModels.h>

class TraubMilesAlt: public NeuronModels::TraubMiles
```

```
public:
    // typedefs

    typedef Snippet::ValueBase<7> ParamValues;
    typedef Models::VarInitContainerBase<4> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::TraubMilesAlt* getInstance();
    virtual std::string getSimCode() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;
    typedef Snippet::ValueBase<7> ParamValues;
    typedef Models::VarInitContainerBase<4> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
    static const NeuronModels::TraubMiles* getInstance();
```

```
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual StringVec getParamNames() const;
    virtual VarVec getVars() const;
```

Detailed Documentation

Hodgkin-Huxley neurons with Traub & Miles algorithm.

Using a workaround to avoid singularity: adding the munimum numerical value of the floating point precision used.

Methods

```
virtual std::string getSimCode() const
```

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name "NN". It needs to refer to the predefined variable "ISYN", i.e. contain , if it is to receive input.

28.6.12 class NeuronModels::TraubMilesFast

Overview

Hodgkin-Huxley neurons with Traub & Miles algorithm: Original fast implementation, using 25 inner iterations. *More...*

```
#include <neuronModels.h>

class TraubMilesFast: public NeuronModels::TraubMiles

public:
    // typedefs

    typedef Snippet::ValueBase<7> ParamValues;
    typedef Models::VarInitContainerBase<4> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::TraubMilesFast* getInstance();
    virtual std::string getSimCode() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
```

```
typedef std::vector<ParamVal> ParamValVec;
typedef std::vector<DerivedParam> DerivedParamVec;
typedef std::vector<Var> VarVec;
typedef Snippet::ValueBase<7> ParamValues;
typedef Models::VarInitContainerBase<4> VarValues;
typedef Models::VarInitContainerBase<0> PreVarValues;
typedef Models::VarInitContainerBase<0> PostVarValues;

// structs

struct DerivedParam;
struct EGP;
struct ParamVal;
struct Var;

// methods

virtual ~Base();
virtual StringVec getParamNames() const;
virtual DerivedParamVec getDerivedParams() const;
virtual VarVec getVars() const;
virtual EGPVec getExtraGlobalParams() const;
size_t getVarIndex(const std::string& varName) const;
size_t getExtraGlobalParamIndex(const std::string& paramName) const;
virtual std::string getSimCode() const;
virtual std::string getThresholdConditionCode() const;
virtual std::string getResetCode() const;
virtual std::string getSupportCode() const;
virtual Models::Base::ParamValVec getAdditionalInputVars() const;
virtual bool isAutoRefractoryRequired() const;
static const NeuronModels::TraubMiles* getInstance();
virtual std::string getSimCode() const;
virtual std::string getThresholdConditionCode() const;
virtual StringVec getParamNames() const;
virtual VarVec getVars() const;
```

Detailed Documentation

Hodgkin-Huxley neurons with Traub & Miles algorithm: Original fast implementation, using 25 inner iterations.

There are singularities in this model, which can be easily hit in float precision

Methods

`virtual std::string getSimCode() const`

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name “NN”. It needs to refer to the predefined variable “ISYN”, i.e. contain , if it is to receive input.

28.6.13 class NeuronModels::TraubMilesNStep

Overview

Hodgkin-Huxley neurons with Traub & Miles algorithm. *More...*

```
#include <neuronModels.h>

class TraubMilesNStep: public NeuronModels::TraubMiles

public:
    // typedefs

    typedef Snippet::ValueBase<8> ParamValues;
    typedef Models::VarInitContainerBase<4> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const NeuronModels::TraubMilesNStep* getInstance();
    virtual std::string getSimCode() const;
    virtual StringVec getParamNames() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;
    typedef Snippet::ValueBase<7> ParamValues;
    typedef Models::VarInitContainerBase<4> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
```

```
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual std::string getResetCode() const;
    virtual std::string getSupportCode() const;
    virtual Models::Base::ParamValVec getAdditionalInputVars() const;
    virtual bool isAutoRefractoryRequired() const;
    static const NeuronModels::TraubMiles* getInstance();
    virtual std::string getSimCode() const;
    virtual std::string getThresholdConditionCode() const;
    virtual StringVec getParamNames() const;
    virtual VarVec getVars() const;
```

Detailed Documentation

Hodgkin-Huxley neurons with Traub & Miles algorithm.

Same as standard *TraubMiles* model but number of inner loops can be set using a parameter

Methods

`virtual std::string getSimCode() const`

Gets the code that defines the execution of one timestep of integration of the neuron model.

The code will refer to for the value of the variable with name "NN". It needs to refer to the predefined variable "ISYN", i.e. contain , if it is to receive input.

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

```
namespace NeuronModels

// classes

class Base;
class Izhikevich;
class IzhikevichVariable;
class LIF;
class Poisson;
class PoissonNew;
class RulkovMap;
class SpikeSource;
class SpikeSourceArray;
class TraubMiles;
class TraubMilesAlt;
class TraubMilesFast;
class TraubMilesNStep;

 // namespace NeuronModels
```

28.7 namespace PostsynapticModels

28.7.1 class PostsynapticModels::Base

Base class for all postsynaptic models.

```
#include <postsynapticModels.h>

class Base: public Models::Base

public:
    // methods

    virtual std::string getDecayCode() const;
    virtual std::string getApplyInputCode() const;
    virtual std::string getSupportCode() const;
;

// direct descendants

class DeltaCurr;
class ExpCond;
class ExpCurr;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
```

28.7.2 class PostsynapticModels::DeltaCurr

Overview

Simple delta current synapse. *More...*

```
#include <postsynapticModels.h>

class DeltaCurr: public PostsynapticModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<0> ParamValues;
    typedef Models::VarInitContainerBase<0> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const DeltaCurr* getInstance();
    virtual std::string getApplyInputCode() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getDecayCode() const;
    virtual std::string getApplyInputCode() const;
```

```
    virtual std::string getSupportCode() const;
```

Detailed Documentation

Simple delta current synapse.

Synaptic input provides a direct inject of instantaneous current

28.7.3 class PostsynapticModels::ExpCond

Overview

Exponential decay with synaptic input treated as a conductance value. *More...*

```
#include <postsynapticModels.h>

class ExpCond: public PostsynapticModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<2> ParamValues;
    typedef Models::VarInitContainerBase<0> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const ExpCond* getInstance();
    virtual std::string getDecayCode() const;
    virtual std::string getApplyInputCode() const;
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;
```

```
    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getDecayCode() const;
    virtual std::string getApplyInputCode() const;
    virtual std::string getSupportCode() const;
```

Detailed Documentation

Exponential decay with synaptic input treated as a conductance value.

This model has no variables and two parameters:

- `tau` : Decay time constant
- `E` : Reversal potential

`tau` is used by the derived parameter `expdecay` which returns expf(-dt/tau).

Methods

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

`virtual DerivedParamVec getDerivedParams() const`

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

28.7.4 class PostsynapticModels::ExpCurr

Overview

Exponential decay with synaptic input treated as a current value. *More...*

```
#include <postsynapticModels.h>

class ExpCurr: public PostsynapticModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<1> ParamValues;
    typedef Models::VarInitContainerBase<0> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;
```

```
    // methods

    static const ExpCurr* getInstance();
    virtual std::string getDecayCode() const;
    virtual std::string getApplyInputCode() const;
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getDecayCode() const;
    virtual std::string getApplyInputCode() const;
    virtual std::string getSupportCode() const;
```

Detailed Documentation

Exponential decay with synaptic input treated as a current value.

Methods

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

`virtual DerivedParamVec getDerivedParams() const`

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

```
namespace PostsynapticModels

// classes

class Base;
class DeltaCurr;
class ExpCond;
class ExpCurr;

 // namespace PostsynapticModels
```

28.8 namespace Snippet

28.8.1 class Snippet::Base

struct Snippet::Base::DerivedParam

A derived parameter has a name and a function for obtaining its value.

```
#include <snippet.h>

struct DerivedParam

    // fields

    std::string name;
    std::function<double(const std::vector<double>&, double)> func;
;
```

struct Snippet::Base::EGP

An extra global parameter has a name and a type.

```
#include <snippet.h>

struct EGP

    // fields

    std::string name;
    std::string type;
;
```

struct Snippet::Base::ParamVal

Additional input variables, row state variables and other things have a name, a type and an initial value.

```
#include <snippet.h>
```

```
struct ParamVal

    // fields

    std::string name;
    std::string type;
    double value;
;
```

Overview

Base class for all code snippets. *More...*

```
#include <snippet.h>

class Base

public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
;

// direct descendants

class Base;
class Base;
class Base;
```

Detailed Documentation

Base class for all code snippets.

Methods

```
virtual StringVec getParamNames() const
```

Gets names of of (independent) model parameters.

```
virtual DerivedParamVec getDerivedParams() const
```

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

28.8.2 template class Snippet::Init

Class used to bind together everything required to utilize a snippet

1. A pointer to a variable initialisation snippet
2. The parameters required to control the variable initialisation snippet

```
#include <snippet.h>

template <typename SnippetBase>
class Init

public:
    // methods

    Init(
        const SnippetBase* snippet,
        const std::vector<double>& params
        );

    const SnippetBase* getSnippet() const;
    const std::vector<double>& getParams() const;
    const std::vector<double>& getDerivedParams() const;
    void initDerivedParams(double dt);
;

// direct descendants

class Init;
class VarInit;
```

28.8.3 template class Snippet::ValueBase<0>

Overview

Template specialisation of *ValueBase* to avoid compiler warnings in the case when a model requires no parameters or state variables *More...*

```
#include <snippet.h>

template <>
class ValueBase<0>

public:
    // methods

    template <typename... T>
    ValueBase(T&&... vals);
```

```
    std::vector<double> getValues() const;
;
```

Detailed Documentation

Template specialisation of *ValueBase* to avoid compiler warnings in the case when a model requires no parameters or state variables

Methods

```
std::vector<double> getValues() const
```

Gets values as a vector of doubles.

28.8.4 template class Snippet::ValueBase

Overview

```
#include <snippet.h>

template <size_t NumVars>
class ValueBase

public:
    // methods

    template <typename... T>
    ValueBase(T&&... vals);

    const std::vector<double>& getValues() const;
    double operator [] (size_t pos) const;
;
```

Detailed Documentation

Methods

```
const std::vector<double>& getValues() const
```

Gets values as a vector of doubles.

Wrapper to ensure at compile time that correct number of values are used when specifying the values of a model's parameters and initial state.

```
namespace Snippet

// classes

class Base;

template <typename SnippetBase>
class Init;
```

```
template <>
class ValueBase<0>;

template <size_t NumVars>
class ValueBase;

 // namespace Snippet
```

28.9 namespace Utils

28.9.1 Overview

```
namespace Utils

// global functions

GENN_EXPORT bool isRNGRequired(const std::string& code);
GENN_EXPORT bool isInitRNGRequired(const std::vector<Models::VarInit>&␣
↪varInitialisers);
GENN_EXPORT bool isTypePointer(const std::string& type);
GENN_EXPORT std::string getUnderlyingType(const std::string& type);

 // namespace Utils
```

28.9.2 Detailed Documentation

Global Functions

`GENN_EXPORT bool isRNGRequired(const std::string& code)`

Does the code string contain any functions requiring random number generator.

`GENN_EXPORT bool isInitRNGRequired(const std::vector<Models::VarInit>&␣ ↪varInitialisers)`

Does the model with the vectors of variable initialisers and modes require an RNG for the specified init location i.e. host or device.

`GENN_EXPORT bool isTypePointer(const std::string& type)`

Function to determine whether a string containing a type is a pointer.

`GENN_EXPORT std::string getUnderlyingType(const std::string& type)`

Assuming type is a string containing a pointer type, function to return the underlying type.

28.10 namespace WeightUpdateModels

28.10.1 class WeightUpdateModels::Base

Overview

Base class for all weight update models. *More...*

```
#include <weightUpdateModels.h>

class Base: public Models::Base

public:
    // methods

    virtual std::string getSimCode() const;
    virtual std::string getEventCode() const;
    virtual std::string getLearnPostCode() const;
    virtual std::string getSynapseDynamicsCode() const;
    virtual std::string getEventThresholdConditionCode() const;
    virtual std::string getSimSupportCode() const;
    virtual std::string getLearnPostSupportCode() const;
    virtual std::string getSynapseDynamicsSuppportCode() const;
    virtual std::string getPreSpikeCode() const;
    virtual std::string getPostSpikeCode() const;
    virtual VarVec getPreVars() const;
    virtual VarVec getPostVars() const;
    virtual bool isPreSpikeTimeRequired() const;
    virtual bool isPostSpikeTimeRequired() const;
    size_t getPreVarIndex(const std::string& varName) const;
    size_t getPostVarIndex(const std::string& varName) const;
;

// direct descendants

class PiecewiseSTDP;
class StaticGraded;
class StaticPulse;
class StaticPulseDendriticDelay;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;
```

```
// methods

virtual ~Base();
virtual StringVec getParamNames() const;
virtual DerivedParamVec getDerivedParams() const;
virtual VarVec getVars() const;
virtual EGPVec getExtraGlobalParams() const;
size_t getVarIndex(const std::string& varName) const;
size_t getExtraGlobalParamIndex(const std::string& paramName) const;
```

Detailed Documentation

Base class for all weight update models.

Methods

`virtual std::string getSimCode() const`

Gets simulation code run when 'true' spikes are received.

`virtual std::string getEventCode() const`

Gets code run when events (all the instances where event threshold condition is met) are received.

`virtual std::string getLearnPostCode() const`

Gets code to include in the learnSynapsesPost kernel/function.

For examples when modelling STDP, this is where the effect of postsynaptic spikes which occur *after* presynaptic spikes are applied.

`virtual std::string getSynapseDynamicsCode() const`

Gets code for synapse dynamics which are independent of spike detection.

`virtual std::string getEventThresholdConditionCode() const`

Gets codes to test for events.

`virtual std::string getSimSupportCode() const`

Gets support code to be made available within the synapse kernel/function.

This is intended to contain user defined device functions that are used in the weight update code. Preprocessor defines are also allowed if appropriately safeguarded against multiple definition by using ifndef; functions should be declared as "__host__ __device__" to be available for both GPU and CPU versions; note that this support code is available to sim, event threshold and event code

`virtual std::string getLearnPostSupportCode() const`

Gets support code to be made available within learnSynapsesPost kernel/function.

Preprocessor defines are also allowed if appropriately safeguarded against multiple definition by using ifndef; functions should be declared as "__host__ __device__" to be available for both GPU and CPU versions.

`virtual std::string getSynapseDynamicsSuppportCode() const`

Gets support code to be made available within the synapse dynamics kernel/function.

Preprocessor defines are also allowed if appropriately safeguarded against multiple definition by using ifndef; functions should be declared as "__host__ __device__" to be available for both GPU and CPU versions.

`virtual std::string getPreSpikeCode() const`

Gets code to be run once per spiking presynaptic neuron before sim code is run on synapses

This is typically for the code to update presynaptic variables. Postsynaptic and synapse variables are not accesible from within this code

`virtual std::string getPostSpikeCode() const`

Gets code to be run once per spiking postsynaptic neuron before learn post code is run on synapses

This is typically for the code to update postsynaptic variables. Presynaptic and synapse variables are not accesible from within this code

`virtual VarVec getPreVars() const`

Gets names and types (as strings) of state variables that are common across all synapses coming from the same presynaptic neuron

`virtual VarVec getPostVars() const`

Gets names and types (as strings) of state variables that are common across all synapses going to the same postsynaptic neuron

`virtual bool isPreSpikeTimeRequired() const`

Whether presynaptic spike times are needed or not.

`virtual bool isPostSpikeTimeRequired() const`

Whether postsynaptic spike times are needed or not.

`size_t getPreVarIndex(const std::string& varName) const`

Find the index of a named presynaptic variable.

`size_t getPostVarIndex(const std::string& varName) const`

Find the index of a named postsynaptic variable.

28.10.2 class WeightUpdateModels::PiecewiseSTDP

Overview

This is a simple STDP rule including a time delay for the finite transmission speed of the synapse. *More...*

```
#include <weightUpdateModels.h>

class PiecewiseSTDP: public WeightUpdateModels::Base

public:
    // methods

    DECLARE_WEIGHT_UPDATE_MODEL(
        PiecewiseSTDP,
        10,
        2,
        0,
        0
        );

    virtual StringVec getParamNames() const;
```

```
    virtual VarVec getVars() const;
    virtual std::string getSimCode() const;
    virtual std::string getLearnPostCode() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual bool isPreSpikeTimeRequired() const;
    virtual bool isPostSpikeTimeRequired() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getEventCode() const;
    virtual std::string getLearnPostCode() const;
    virtual std::string getSynapseDynamicsCode() const;
    virtual std::string getEventThresholdConditionCode() const;
    virtual std::string getSimSupportCode() const;
    virtual std::string getLearnPostSupportCode() const;
    virtual std::string getSynapseDynamicsSuppportCode() const;
    virtual std::string getPreSpikeCode() const;
    virtual std::string getPostSpikeCode() const;
    virtual VarVec getPreVars() const;
    virtual VarVec getPostVars() const;
    virtual bool isPreSpikeTimeRequired() const;
    virtual bool isPostSpikeTimeRequired() const;
    size_t getPreVarIndex(const std::string& varName) const;
    size_t getPostVarIndex(const std::string& varName) const;
```

Detailed Documentation

This is a simple STDP rule including a time delay for the finite transmission speed of the synapse.

The STDP window is defined as a piecewise function:

The STDP curve is applied to the raw synaptic conductance `gRaw`, which is then filtered through the sugmoidal filter displayed above to obtain the value of `g`.

The STDP curve implies that unpaired pre- and post-synaptic spikes incur a negative increment in `gRaw` (and hence in `g`).

The time of the last spike in each neuron, "sTXX", where XX is the name of a neuron population is (somewhat arbitrarily) initialised to -10.0 ms. If neurons never spike, these spike times are used.

It is the raw synaptic conductance `gRaw` that is subject to the STDP rule. The resulting synaptic conductance is a sigmoid filter of `gRaw`. This implies that `g` is initialised but not `gRaw`, the synapse will revert to the value that corresponds to `gRaw`.

An example how to use this synapse correctly is given in `map_classol.cc` (MBody1 userproject):

```
for (int i= 0; i < model.neuronN[1]*model.neuronN[3]; i++)
        if (gKCDN[i] < 2.0*SCALAR_MIN)
            cnt++;
            fprintf(stdout, "Too low conductance value %e detected and set to␣
↪2*SCALAR_MIN= %e, at index %d ", gKCDN[i], 2*SCALAR_MIN, i);
            gKCDN[i] = 2.0*SCALAR_MIN; //to avoid log(0)/0 below

        scalar tmp = gKCDN[i] / myKCDN_p[5]*2.0 ;
        gRawKCDN[i]=  0.5 * log( tmp / (2.0 - tmp)) /myKCDN_p[7] +␣
↪myKCDN_p[6];

cerr << "Total number of low value corrections: " << cnt << endl;
```

One cannot set values of `g` fully to `0`, as this leads to `gRaw` = -infinity and this is not support. I.e., 'g' needs to be some nominal value > 0 (but can be extremely small so that it acts like it's 0).

The model has 2 variables:

- `g`: conductance of `scalar` type
- `gRaw`: raw conductance of `scalar` type

Parameters are (compare to the figure above):

- `tLrn`: Time scale of learning changes

- `tChng`: Width of learning window
- `tDecay`: Time scale of synaptic strength decay
- `tPunish10`: Time window of suppression in response to 1/0
- `tPunish01`: Time window of suppression in response to 0/1
- `gMax`: Maximal conductance achievable
- `gMid`: Midpoint of sigmoid g filter curve
- `gSlope`: Slope of sigmoid g filter curve
- `tauShift`: Shift of learning curve
- `gSyn0`: Value of syn conductance g decays to

Methods

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

`virtual VarVec getVars() const`

Gets names and types (as strings) of model variables.

`virtual std::string getSimCode() const`

Gets simulation code run when 'true' spikes are received.

`virtual std::string getLearnPostCode() const`

Gets code to include in the learnSynapsesPost kernel/function.

For examples when modelling STDP, this is where the effect of postsynaptic spikes which occur *after* presynaptic spikes are applied.

`virtual DerivedParamVec getDerivedParams() const`

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

`virtual bool isPreSpikeTimeRequired() const`

Whether presynaptic spike times are needed or not.

`virtual bool isPostSpikeTimeRequired() const`

Whether postsynaptic spike times are needed or not.

28.10.3 class WeightUpdateModels::StaticGraded

Overview

Graded-potential, static synapse. *More...*

```
#include <weightUpdateModels.h>

class StaticGraded: public WeightUpdateModels::Base

public:
```

```
    // methods

    DECLARE_WEIGHT_UPDATE_MODEL(
        StaticGraded,
        2,
        1,
        0,
        0
        );

    virtual StringVec getParamNames() const;
    virtual VarVec getVars() const;
    virtual std::string getEventCode() const;
    virtual std::string getEventThresholdConditionCode() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getEventCode() const;
    virtual std::string getLearnPostCode() const;
    virtual std::string getSynapseDynamicsCode() const;
    virtual std::string getEventThresholdConditionCode() const;
    virtual std::string getSimSupportCode() const;
    virtual std::string getLearnPostSupportCode() const;
    virtual std::string getSynapseDynamicsSuppportCode() const;
    virtual std::string getPreSpikeCode() const;
    virtual std::string getPostSpikeCode() const;
```

```
virtual VarVec getPreVars() const;
virtual VarVec getPostVars() const;
virtual bool isPreSpikeTimeRequired() const;
virtual bool isPostSpikeTimeRequired() const;
size_t getPreVarIndex(const std::string& varName) const;
size_t getPostVarIndex(const std::string& varName) const;
```

Detailed Documentation

Graded-potential, static synapse.

In a graded synapse, the conductance is updated gradually with the rule:

$$gSyn = g * tanh((V - E_{pre})/V_{slope}$$

whenever the membrane potential V is larger than the threshold E_{pre}. The model has 1 variable:

- `g`: conductance of `scalar` type

The parameters are:

- `Epre`: Presynaptic threshold potential
- `Vslope`: Activation slope of graded release

`event` code is:

```
$(addToInSyn, $(g)* tanh(($(V_pre)-($(Epre)))*DT*2/$(Vslope)));
```

`event` threshold condition code is:

```
$(V_pre) > $(Epre)
```

The pre-synaptic variables are referenced with the suffix `_pre` in synapse related code such as an the event threshold test. Users can also access post-synaptic neuron variables using the suffix `_post`.

Methods

`virtual StringVec getParamNames() const`

Gets names of of (independent) model parameters.

`virtual VarVec getVars() const`

Gets names and types (as strings) of model variables.

`virtual std::string getEventCode() const`

Gets code run when events (all the instances where event threshold condition is met) are received.

`virtual std::string getEventThresholdConditionCode() const`

Gets codes to test for events.

28.10.4 class WeightUpdateModels::StaticPulse

Overview

Pulse-coupled, static synapse. *More...*

```
#include <weightUpdateModels.h>

class StaticPulse: public WeightUpdateModels::Base

public:
    // methods

    DECLARE_WEIGHT_UPDATE_MODEL(
        StaticPulse,
        0,
        1,
        0,
        0
        );

    virtual VarVec getVars() const;
    virtual std::string getSimCode() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getEventCode() const;
    virtual std::string getLearnPostCode() const;
    virtual std::string getSynapseDynamicsCode() const;
    virtual std::string getEventThresholdConditionCode() const;
    virtual std::string getSimSupportCode() const;
    virtual std::string getLearnPostSupportCode() const;
```

```
    virtual std::string getSynapseDynamicsSuppportCode() const;
    virtual std::string getPreSpikeCode() const;
    virtual std::string getPostSpikeCode() const;
    virtual VarVec getPreVars() const;
    virtual VarVec getPostVars() const;
    virtual bool isPreSpikeTimeRequired() const;
    virtual bool isPostSpikeTimeRequired() const;
    size_t getPreVarIndex(const std::string& varName) const;
    size_t getPostVarIndex(const std::string& varName) const;
```

Detailed Documentation

Pulse-coupled, static synapse.

No learning rule is applied to the synapse and for each pre-synaptic spikes, the synaptic conductances are simply added to the postsynaptic input variable. The model has 1 variable:

- g - conductance of scalar type and no other parameters.

`sim` code is:

```
"$(addToInSyn, $(g));"
```

Methods

`virtual VarVec getVars() const`

Gets names and types (as strings) of model variables.

`virtual std::string getSimCode() const`

Gets simulation code run when 'true' spikes are received.

28.10.5 class WeightUpdateModels::StaticPulseDendriticDelay

Overview

Pulse-coupled, static synapse with heterogenous dendritic delays. *More...*

```
#include <weightUpdateModels.h>

class StaticPulseDendriticDelay: public WeightUpdateModels::Base

public:
    // typedefs

    typedef Snippet::ValueBase<0> ParamValues;
    typedef Models::VarInitContainerBase<2> VarValues;
    typedef Models::VarInitContainerBase<0> PreVarValues;
    typedef Models::VarInitContainerBase<0> PostVarValues;

    // methods

    static const StaticPulseDendriticDelay* getInstance();
    virtual VarVec getVars() const;
```

```
    virtual std::string getSimCode() const;
;
```

Inherited Members

```
public:
    // typedefs

    typedef std::vector<std::string> StringVec;
    typedef std::vector<EGP> EGPVec;
    typedef std::vector<ParamVal> ParamValVec;
    typedef std::vector<DerivedParam> DerivedParamVec;
    typedef std::vector<Var> VarVec;

    // structs

    struct DerivedParam;
    struct EGP;
    struct ParamVal;
    struct Var;

    // methods

    virtual ~Base();
    virtual StringVec getParamNames() const;
    virtual DerivedParamVec getDerivedParams() const;
    virtual VarVec getVars() const;
    virtual EGPVec getExtraGlobalParams() const;
    size_t getVarIndex(const std::string& varName) const;
    size_t getExtraGlobalParamIndex(const std::string& paramName) const;
    virtual std::string getSimCode() const;
    virtual std::string getEventCode() const;
    virtual std::string getLearnPostCode() const;
    virtual std::string getSynapseDynamicsCode() const;
    virtual std::string getEventThresholdConditionCode() const;
    virtual std::string getSimSupportCode() const;
    virtual std::string getLearnPostSupportCode() const;
    virtual std::string getSynapseDynamicsSuppportCode() const;
    virtual std::string getPreSpikeCode() const;
    virtual std::string getPostSpikeCode() const;
    virtual VarVec getPreVars() const;
    virtual VarVec getPostVars() const;
    virtual bool isPreSpikeTimeRequired() const;
    virtual bool isPostSpikeTimeRequired() const;
    size_t getPreVarIndex(const std::string& varName) const;
    size_t getPostVarIndex(const std::string& varName) const;
```

Detailed Documentation

Pulse-coupled, static synapse with heterogenous dendritic delays.

No learning rule is applied to the synapse and for each pre-synaptic spikes, the synaptic conductances are simply added to the postsynaptic input variable. The model has 2 variables:

- g - conductance of scalar type
- d - dendritic delay in timesteps and no other parameters.

`sim` code is:

```
" $(addToInSynDelay, $(g), $(d));
```

Methods

```
virtual VarVec getVars() const
```

Gets names and types (as strings) of model variables.

```
virtual std::string getSimCode() const
```

Gets simulation code run when 'true' spikes are received.

```
namespace WeightUpdateModels

// classes

class Base;
class PiecewiseSTDP;
class StaticGraded;
class StaticPulse;
class StaticPulseDendriticDelay;

 // namespace WeightUpdateModels
```

28.11 namespace filesystem

28.12 namespace pygenn

28.12.1 namespace pygenn::genn_groups

class pygenn::genn_groups::CurrentSource

Overview

Class representing a current injection into a group of neurons. *More...*

```
class CurrentSource: public pygenn::genn_groups::Group

public:
    // fields

     current_source_model;
     target_pop;
     pop;

    // methods

    def __init__();
```

```
    def size();
    def set_current_source_model();
    def add_to();
    def add_extra_global_param();
    def load();
    def reinitialise();
;
```

Inherited Members

```
public:
    // fields

     name;
     vars;
     extra_global_params;

    // methods

    def __init__();
    def set_var();
```

Detailed Documentation

Class representing a current injection into a group of neurons.

Methods

`def __init__()`

Init *CurrentSource*.

Parameters:

name	string name of the current source

`def size()`

Number of neuron in the injected population.

`def set_current_source_model()`

Set curront source model, its parameters and initial variables.

Parameters:

model	type as string of intance of the model
param_space	dict with model parameters
var_space	dict with model variables

`def add_to()`

Inject this *CurrentSource* into population and add it to the GeNN NNmodel.

Parameters:

pop	instance of *NeuronGroup* into which this *CurrentSource* should be injected
nn_model	GeNN NNmodel

`def add_extra_global_param()`

Add extra global parameter.

Parameters:

param_name	string with the name of the extra global parameter
param_values	iterable or a single value

`def reinitialise()`

Reinitialise current source.

Parameters:

slm	SharedLibraryModel instance for acccessing variables
scalar	String specifying "scalar" type

class pygenn::genn_groups::Group

Overview

Parent class of *NeuronGroup*, *SynapseGroup* and *CurrentSource*. *More...*

```
class Group: public object

public:
    // fields

     name;
     vars;
     extra_global_params;

    // methods

    def __init__();
    def set_var();
;

// direct descendants
```

```
class CurrentSource;
class NeuronGroup;
class SynapseGroup;
```

Detailed Documentation

Parent class of *NeuronGroup*, *SynapseGroup* and *CurrentSource*.

Methods

```
def __init__()
```

Init *Group*.

Parameters:

name	string name of the *Group*

```
def set_var()
```

Set values for a Variable.

Parameters:

var_name	string with the name of the variable
values	iterable or a single value

class pygenn::genn_groups::NeuronGroup

Overview

Class representing a group of neurons. *More...*

```
class NeuronGroup: public pygenn::genn_groups::Group

public:
    // fields

     neuron;
     spikes;
     spike_count;
     spike_que_ptr;
     is_spike_source_array;
     type;
     pop;

    // methods
```

```
    def __init__();
    def current_spikes();
    def delay_slots();
    def size();
    def set_neuron();
    def add_to();
    def add_extra_global_param();
    def load();
    def reinitialise();
;
```

Inherited Members

```
public:
    // fields

     name;
     vars;
     extra_global_params;

    // methods

    def __init__();
    def set_var();
```

Detailed Documentation

Class representing a group of neurons.

Methods

`def __init__()`

Init *NeuronGroup*.

Parameters:

name	string name of the group

`def current_spikes()`

Current spikes from GeNN.

`def delay_slots()`

Maximum delay steps needed for this group.

`def set_neuron()`

Set neuron, its parameters and initial variables.

Parameters:

model	type as string of intance of the model
param_space	dict with model parameters
var_space	dict with model variables

`def add_to()`

Add this *NeuronGroup* to a model.

Parameters:

model_spec	`pygenn.genn_model.GeNNModel` to add to
num_neurons	int number of neurons

`def add_extra_global_param()`

Add extra global parameter.

Parameters:

param_name	string with the name of the extra global parameter
param_values	iterable or a single value

`def load()`

Loads neuron group.

Parameters:

slm	SharedLibraryModel instance for acccessing variables
scalar	String specifying “scalar” type

`def reinitialise()`

Reinitialise neuron group.

Parameters:

slm	SharedLibraryModel instance for acccessing variables
scalar	String specifying “scalar” type

class pygenn::genn_groups::SynapseGroup

Overview

Class representing synaptic connection between two groups of neurons. *More...*

```
class SynapseGroup: public pygenn::genn_groups::Group

public:
    // fields

     connections_set;
     w_update;
     postsyn;
     src;
     trg;
     psm_vars;
     pre_vars;
     post_vars;
     connectivity_initialiser;
     synapse_order;
     ind;
     row_lengths;
     pop;

    // methods

    def __init__();
    def num_synapses();
    def weight_update_var_size();
    def max_row_length();
    def set_psm_var();
    def set_pre_var();
    def set_post_var();
    def set_weight_update();
    def set_post_syn();
    def get_var_values();
    def is_connectivity_init_required();
    def matrix_type();
    def is_ragged();
    def is_bitmask();
    def is_dense();
    def has_individual_synapse_vars();
    def has_individual_postsynaptic_vars();
    def set_sparse_connections();
    def get_sparse_pre_inds();
    def get_sparse_post_inds();
    def set_connected_populations();
    def add_to();
    def add_extra_global_param();
    def load();
    def reinitialise();
;
```

Inherited Members

```
public:
    // fields

     name;
     vars;
     extra_global_params;

    // methods

    def __init__();
    def set_var();
```

Detailed Documentation

Class representing synaptic connection between two groups of neurons.

Methods

`def __init__()`

Init *SynapseGroup*.

Parameters:

name	string name of the group

`def num_synapses()`

Number of synapses in group.

`def weight_update_var_size()`

Size of each weight update variable.

`def set_psm_var()`

Set values for a postsynaptic model variable.

Parameters:

var_name	string with the name of the postsynaptic model variable
values	iterable or a single value

`def set_pre_var()`

Set values for a presynaptic variable.

Parameters:

var_name	string with the name of the presynaptic variable
values	iterable or a single value

`def set_post_var()`

Set values for a postsynaptic variable.

Parameters:

var_name	string with the name of the presynaptic variable
values	iterable or a single value

`def set_weight_update()`

Set weight update model, its parameters and initial variables.

Parameters:

model	type as string of intance of the model
param_space	dict with model parameters
var_space	dict with model variables
pre_var_space	dict with model presynaptic variables
post_var_space	dict with model postsynaptic variables

`def set_post_syn()`

Set postsynaptic model, its parameters and initial variables.

Parameters:

model	type as string of intance of the model
param_space	dict with model parameters
var_space	dict with model variables

`def matrix_type()`

Type of the projection matrix.

`def is_ragged()`

Tests whether synaptic connectivity uses Ragged format.

`def is_bitmask()`

Tests whether synaptic connectivity uses Bitmask format.

`def is_dense()`

Tests whether synaptic connectivity uses dense format.

`def has_individual_synapse_vars()`

Tests whether synaptic connectivity has individual weights.

`def has_individual_postsynaptic_vars()`

Tests whether synaptic connectivity has individual postsynaptic model variables.

`def set_sparse_connections()`

Set ragged format connections between two groups of neurons.

Parameters:

pre_indices	ndarray of presynaptic indices
post_indices	ndarray of postsynaptic indices

`def get_sparse_pre_inds()`

Get presynaptic indices of synapse group connections.

Returns:

ndarray of presynaptic indices

`def get_sparse_post_inds()`

Get postsynaptic indices of synapse group connections.

Returns:

ndarrays of postsynaptic indices

`def set_connected_populations()`

Set two groups of neurons connected by this *SynapseGroup*.

Parameters:

source	string name of the presynaptic neuron group
target	string name of the postsynaptic neuron group

`def add_to()`

Add this *SynapseGroup* to the a model.

Parameters:

model_spec	`pygenn.genn_model.GeNNModel` to add to
delay_steps	number of axonal delay timesteps to simulate for this synapse group

`def add_extra_global_param()`

Add extra global parameter.

Parameters:

param_name	string with the name of the extra global parameter
param_values	iterable or a single value

```
def reinitialise()
```

Reinitialise synapse group.

Parameters:

slm	SharedLibraryModel instance for acccessing variables
scalar	String specifying “scalar” type

Overview

```
namespace genn_groups

// classes

class CurrentSource;
class Group;
class NeuronGroup;
class SynapseGroup;

// global variables

 xrange;

 // namespace genn_groups
```

Detailed Documentation

Global Variables

```
 xrange
```

GeNNGroups This module provides classes which automatize model checks and parameter convesions for GeNN Groups.

28.12.2 namespace pygenn::genn_model

class pygenn::genn_model::GeNNModel

Overview

GeNNModel class This class helps to define, build and run a GeNN model from python. *More...*

```
class GeNNModel: public object

public:
    // fields

     use_backend;
     default_var_location;
     model_name;
     neuron_populations;
     synapse_populations;
     current_sources;
     dT;
     T;

    // methods

    def __init__();
    def use_backend();
    def default_var_location();
    def default_sparse_connectivity_location();
    def model_name();
    def t();
    def timestep();
    def dT();
    def add_neuron_population();
    def add_synapse_population();
    def add_current_source();
    def build();
    def load();
    def reinitialise();
    def step_time();
    def pull_state_from_device();
    def pull_spikes_from_device();
    def pull_current_spikes_from_device();
    def pull_connectivity_from_device();
    def pull_var_from_device();
    def push_state_to_device();
    def push_spikes_to_device();
    def push_current_spikes_to_device();
    def push_connectivity_to_device();
    def push_var_to_device();
    def end();
;
```

Detailed Documentation

GeNNModel class This class helps to define, build and run a GeNN model from python.

Methods

```
def __init__()
```

Init *GeNNModel*.

Parameters:

precision	string precision as string (“float”, “double” or “long double”). defaults to float.
model_name	string name of the model. Defaults to “GeNNModel”.
enable_debug	boolean enable debug mode. Disabled by default.
backend	string specifying name of backend module to use Defaults to None to pick ‘best’ backend for your system

`def default_var_location()`

Default variable location - defines where state variables are initialised.

`def default_sparse_connectivity_location()`

Default sparse connectivity mode - where connectivity is initialised.

`def model_name()`

Name of the model.

`def t()`

Simulation time in ms.

`def timestep()`

Simulation time step.

`def dT()`

Step size.

`def add_neuron_population()`

Add a neuron population to the GeNN model.

Parameters:

pop_name	name of the new population
num_neurons	number of neurons in the new population
neuron	type of the *NeuronModels* class as string or instance of neuron class derived from `pygenn.genn_wrapper.NeuronModels.Custom` (see also *pygenn.genn_model.create_custom_neuron_class*)
param_space	dict with param values for the *NeuronModels* class
var_space	dict with initial variable values for the *NeuronModels* class

`def add_synapse_population()`

Add a synapse population to the GeNN model.

Parameters:

pop_name	name of the new population
matrix_type	type of the matrix as string
delay_steps	delay in number of steps
source	source neuron group
target	target neuron group
w_update_model	type of the *WeightUpdateModels* class as string or instance of weight update model class derived from `pygenn.genn_wrapper.WeightUpdateModels.Custom` (see also *pygenn.genn_model.create_custom_weight_update_class*)
wu_param_space	dict with param values for the *WeightUpdateModels* class
wu_var_space	dict with initial values for *WeightUpdateModels* state variables
wu_pre_var_space	dict with initial values for *WeightUpdateModels* presynaptic variables
wu_post_var_space	dict with initial values for *WeightUpdateModels* postsynaptic variables
postsyn_model	type of the *PostsynapticModels* class as string or instance of postsynaptic model class derived from `pygenn.genn_wrapper.PostsynapticModels.Custom` (see also *pygenn.genn_model.create_custom_postsynaptic_class*)
ps_param_space	dict with param values for the *PostsynapticModels* class
ps_var_space	dict with initial variable values for the *PostsynapticModels* class
connectivity_initialiser	*InitSparseConnectivitySnippet::Init* for connectivity

`def add_current_source()`

Add a current source to the GeNN model.

Parameters:

cs_name	name of the new current source
current_source_model	type of the *CurrentSourceModels* class as string or instance of *CurrentSourceModels* class derived from `pygenn.genn_wrapper.CurrentSourceModels.Custom` (see also *pygenn.genn_model.create_custom_current_source_class*)
pop_name	name of the population into which the current source should be injected
param_space	dict with param values for the *CurrentSourceModels* class
var_space	dict with initial variable values for the *CurrentSourceModels* class

`def build()`

Finalize and build a GeNN model.

Parameters:

path_to_model	path where to place the generated model code. Defaults to the local directory.

`def load()`

import the model as shared library and initialize it

`def reinitialise()`

reinitialise model to its original state without re-loading

`def pull_state_from_device()`

Pull state from the device for a given population.

```
def pull_spikes_from_device()
```

Pull spikes from the device for a given population.

```
def pull_current_spikes_from_device()
```

Pull spikes from the device for a given population.

```
def pull_connectivity_from_device()
```

Pull connectivity from the device for a given population.

```
def pull_var_from_device()
```

Pull variable from the device for a given population.

```
def push_state_to_device()
```

Push state to the device for a given population.

```
def push_spikes_to_device()
```

Push spikes to the device for a given population.

```
def push_current_spikes_to_device()
```

Push current spikes to the device for a given population.

```
def push_connectivity_to_device()
```

Push connectivity to the device for a given population.

```
def push_var_to_device()
```

Push variable to the device for a given population.

```
def end()
```

Free memory.

Overview

```
namespace genn_model

// classes

class GeNNModel;

// global variables

tuple backend_modules;
tuple m;

// global functions

def init_var();
def init_connectivity();
def create_custom_neuron_class();
def create_custom_postsynaptic_class();
def create_custom_weight_update_class();
def create_custom_current_source_class();
```

```
def create_custom_model_class();
def create_dpf_class();
def create_cmlf_class();
def create_custom_init_var_snippet_class();
def create_custom_sparse_connect_init_snippet_class();

 // namespace genn_model
```

Detailed Documentation

Global Functions

```
def init_var()
```

This helper function creates a VarInit object to easily initialise a variable using a snippet.

Parameters:

init_var_snippet	type of the *InitVarSnippet* class as string or instance of class derived from InitVarSnippet::Custom class.
param_space	dict with param values for the *InitVarSnippet* class

```
def init_connectivity()
```

This helper function creates a *InitSparseConnectivitySnippet::Init* object to easily initialise connectivity using a snippet.

Parameters:

init_sparse_connect_snippet	type of the *InitSparseConnectivitySnippet* class as string or instance of class derived from InitSparseConnectivitySnippet::Custom.
param_space	dict with param values for the *InitSparseConnectivitySnippet* class

```
def create_custom_neuron_class()
```

This helper function creates a custom NeuronModel class.

Parameters:

class_name	name of the new class
param_names	list of strings with param names of the model
var_name_types	list of pairs of strings with varible names and types of the model
derived_params	list of pairs, where the first member is string with name of the derived parameter and the second MUST be an instance of a class which inherits from ``pygenn.genn_wrapper.Snippet.DerivedParamFunc`` @param sim_code string with the simulation code @param threshold_condition_code string with the threshold condition code @param reset_code string with the reset code @param support_code string with the support code @param extra_global_params list of pairs of strings with names and types of additional parameters
additional_input_vars	list of tuples with names and types as strings and initial values of additional local input variables
is_auto_refractory_required	does this model require auto-refractory logic to be generated?
custom_body	dictionary with additional attributes and methods of the new class

See also:

create_custom_postsynaptic_class

create_custom_weight_update_class

create_custom_current_source_class

create_custom_init_var_snippet_class

create_custom_sparse_connect_init_snippet_class

`def create_custom_postsynaptic_class()`

This helper function creates a custom PostsynapticModel class.

Parameters:

class_name	name of the new class
param_names	list of strings with param names of the model
var_name_types	list of pairs of strings with varible names and types of the model
derived_params	list of pairs, where the first member is string with name of the derived parameter and the second MUST be an instance of a class which inherits from `pygenn.genn_wrapper.DerivedParamFunc`
decay_code	string with the decay code
apply_input_code	string with the apply input code
support_code	string with the support code
custom_body	dictionary with additional attributes and methods of the new class

See also:

create_custom_neuron_class

create_custom_weight_update_class

create_custom_current_source_class

create_custom_init_var_snippet_class

create_custom_sparse_connect_init_snippet_class

```
def create_custom_weight_update_class()
```

This helper function creates a custom WeightUpdateModel class.

Parameters:

class_name	name of the new class
param_names	list of strings with param names of the model
var_name_types	list of pairs of strings with variable names and types of the model
pre_var_name_types	list of pairs of strings with presynaptic variable names and types of the model
post_var_name_types	list of pairs of strings with postsynaptic variable names and types of the model
derived_params	list of pairs, where the first member is string with name of the derived parameter and the second MUST be an instance of a class which inherits from ``pygenn.genn_wrapper.DerivedParamFunc`` @param sim_code string with the simulation code @param event_code string with the event code @param learn_post_code string with the code to include in learn_synapse_post kernel/function
synapse_dynamics_code	string with the synapse dynamics code
event_threshold_condition_code	string with the event threshold condition code
pre_spike_code	string with the code run once per spiking presynaptic neuron
post_spike_code	string with the code run once per spiking postsynaptic neuron
sim_support_code	string with simulation support code
learn_post_support_code	string with support code for learn_synapse_post kernel/function
synapse_dynamics_support_code	string with the synapse dynamics support code
extra_global_params	list of pairs of strings with names and types of additional parameters
is_pre_spike_time_required	boolean, is presynaptic spike time required in any weight update kernels?
is_post_spike_time_required	boolean, is postsynaptic spike time required in any weight update kernels?
custom_body	dictionary with additional attributes and methods of the new class

See also:

create_custom_neuron_class

create_custom_postsynaptic_class

create_custom_current_source_class

create_custom_init_var_snippet_class

create_custom_sparse_connect_init_snippet_class

`def create_custom_current_source_class()`

This helper function creates a custom NeuronModel class.

Parameters:

class_name	name of the new class
param_names	list of strings with param names of the model
var_name_types	list of pairs of strings with varible names and types of the model
derived_params	list of pairs, where the first member is string with name of the derived parameter and the second MUST be an instance of the class which inherits from `pygenn.genn_wrapper.DerivedParamFunc`
injection_code	string with the current injection code
extra_global_params	list of pairs of strings with names and types of additional parameters
custom_body	dictionary with additional attributes and methods of the new class

See also:

create_custom_neuron_class

create_custom_weight_update_class

create_custom_current_source_class

create_custom_init_var_snippet_class

create_custom_sparse_connect_init_snippet_class

`def create_custom_model_class()`

This helper function completes a custom model class creation.

This part is common for all model classes and is nearly useless on its own unless you specify custom_body.

Parameters:

class_name	name of the new class
base	base class
param_names	list of strings with param names of the model
var_name_types	list of pairs of strings with varible names and types of the model
derived_params	list of pairs, where the first member is string with name of the derived parameter and the second MUST be an instance of the class which inherits from the `pygenn.genn_wrapper.DerivedParamFunc` class
custom_body	dictionary with attributes and methods of the new class

See also:

create_custom_neuron_class

create_custom_weight_update_class

create_custom_postsynaptic_class

create_custom_current_source_class

create_custom_init_var_snippet_class

create_custom_sparse_connect_init_snippet_class

`def create_dpf_class()`

Helper function to create derived parameter function class.

Parameters:

dp_func	a function which computes the derived parameter and takes two args "pars" (vector of double) and "dt" (double)

`def create_cmlf_class()`

Helper function to create function class for calculating sizes of matrices initialised with sparse connectivity initialisation snippet.

Parameters:

cml_func	a function which computes the length and takes three args "num_pre" (unsigned int), "num_post" (unsigned int) and "pars" (vector of double)

`def create_custom_init_var_snippet_class()`

This helper function creates a custom *InitVarSnippet* class.

Parameters:

class_name	name of the new class
param_names	list of strings with param names of the model
derived_params	list of pairs, where the first member is string with name of the derived parameter and the second MUST be an instance of the `pygenn.genn_wrapper.DerivedParamFunc` ' class
var_init_code	string with the variable initialization code
custom_body	dictionary with additional attributes and methods of the new class

See also:

create_custom_neuron_class

create_custom_weight_update_class

create_custom_postsynaptic_class

create_custom_current_source_class

create_custom_sparse_connect_init_snippet_class

`def create_custom_sparse_connect_init_snippet_class()`

This helper function creates a custom *InitSparseConnectivitySnippet* class.

Parameters:

class_name	name of the new class
param_names	list of strings with param names of the model
derived_params	list of pairs, where the first member is string with name of the derived parameter and the second MUST be an instance of the class which inherits from `pygenn.genn_wrapper.DerivedParamFunc`
row_build_code	string with row building initialization code
row_build_state_vars	list of tuples of state variables, their types and their initial values to use across row building loop
calc_max_row_len_func	instance of class inheriting from CalcMaxLengthFunc used to calculate maximum row length of synaptic matrix
calc_max_col_len_func	instance of class inheriting from CalcMaxLengthFunc used to calculate maximum col length of synaptic matrix
extra_global_params	list of pairs of strings with names and types of additional parameters
custom_body	dictionary with additional attributes and methods of the new class

See also:

create_custom_neuron_class

create_custom_weight_update_class

create_custom_postsynaptic_class

create_custom_current_source_class

create_custom_init_var_snippet_class

28.12.3 namespace pygenn::model_preprocessor

class pygenn::model_preprocessor::ExtraGlobalVariable

Overview

Class holding information about GeNN extra global pointer variable. *More...*

```
class ExtraGlobalVariable: public object

public:
    // fields

     name;
     type;
     view;
     values;

    // methods

    def __init__();
    def set_values();
;
```

Detailed Documentation

Class holding information about GeNN extra global pointer variable.

Methods

```
def __init__()
```

Init *Variable*.

Parameters:

variable_name	string name of the variable
variable_type	string type of the variable
values	iterable

```
def set_values()
```

Set *Variable* 's values.

Parameters:

values	iterable, single value or VarInit instance

class pygenn::model_preprocessor::Variable

Overview

Class holding information about GeNN variables. *More...*

```
class Variable: public object

public:
    // fields

     name;
     type;
     view;
     needs_allocation;
     init_required;
     init_val;
     values;

    // methods

    def __init__();
    def set_values();
;
```

Detailed Documentation

Class holding information about GeNN variables.

Methods

```
def __init__()
```

Init *Variable*.

Parameters:

variable_name	string name of the variable
variable_type	string type of the variable
values	iterable, single value or VarInit instance

```
def set_values()
```

Set *Variable* 's values.

Parameters:

values	iterable, single value or VarInit instance

Overview

```
namespace model_preprocessor

// classes

class ExtraGlobalVariable;
class Variable;

// global variables

dictionary genn_to_numpy_types;

// global functions

def prepare_model();
def prepare_snippet();
def is_model_valid();
def param_space_to_vals();
def param_space_to_val_vec();
def var_space_to_vals();
def pre_var_space_to_vals();
def post_var_space_to_vals();

 // namespace model_preprocessor
```

Detailed Documentation

Global Functions

`def prepare_model()`

Prepare a model by checking its validity and extracting information about variables and parameters.

Parameters:

model	string or instance of a class derived from `pygenn.genn_wrapper.NeuronModels.Custom` or `pygenn.genn_wrapper.WeightUpdateModels.Custom` or ``pygenn.genn_wrapper.CurrentSourceModels.Custom`` @param param_space dict with model parameters @param var_space dict with model variables @param pre_var_space optional dict with (weight update) model presynaptic variables
post_var_space	optional dict with (weight update) model postsynaptic variables
model_family	`pygenn.genn_wrapper.NeuronModels` or `pygenn.genn_wrapper.WeightUpdateModels` or ``pygenn.genn_wrapper.CurrentSourceModels`` @return tuple consisting of (model instance, model type, model parameter names, model parameters, list of variable names, dict mapping names of variables to instances of class *Variable*)

`def prepare_snippet()`

Prepare a snippet by checking its validity and extracting information about parameters.

Parameters:

snippet	string or instance of a class derived from `pygenn.genn_wrapper.InitVarSnippet.Custom` or ``pygenn.genn_wrapper.InitSparseConnectivitySnippet.Custom`` @param param_space dict with model parameters @param snippet_family ``pygenn.genn_wrapper.InitVarSnippet`` or ``pygenn.genn_wrapper.InitSparseConnectivitySnippet`` @return tuple consisting of (snippet instance, snippet type, snippet parameter names, snippet parameters)

`def is_model_valid()`

Check whether the model is valid, i.e is native or derived from model_family.Custom.

Raises ValueError if model is not valid (i.e. is not custom and is not natively available)

Parameters:

model	string or instance of model_family.Custom
model_family	model family (*NeuronModels*, *WeightUpdateModels* or *PostsynapticModels*) to which model should belong to

Returns:

instance of the model and its type as string

`def param_space_to_vals()`

Convert a param_space dict to ParamValues.

Parameters:

model	instance of the model
param_space	dict with parameters

Returns:

native model's ParamValues

`def param_space_to_val_vec()`

Convert a param_space dict to a std::vector<double>

Parameters:

model	instance of the model
param_space	dict with parameters

Returns:

native vector of parameters

`def var_space_to_vals()`

Convert a var_space dict to VarValues.

Parameters:

model	instance of the model
var_space	dict with Variables

Returns:

native model's VarValues

```
def pre_var_space_to_vals()
```

Convert a var_space dict to PreVarValues.

Parameters:

model	instance of the weight update model
var_space	dict with Variables

Returns:

native model's VarValues

```
def post_var_space_to_vals()
```

Convert a var_space dict to PostVarValues.

Parameters:

model	instance of the weight update model
var_space	dict with Variables

Returns:

native model's VarValues

```
namespace pygenn

// namespaces

namespace pygenn::genn_groups;
namespace pygenn::genn_model;
namespace pygenn::model_preprocessor;

 // namespace pygenn
```

28.13 namespace std

STL namespace.

28.14 enum FloatType

Floating point precision to use for models.

```
#include <modelSpec.h>

enum FloatType

    GENN_DOUBLE,
    GENN_LONG_DOUBLE,
;
```

28.15 enum MathsFunc

```
enum MathsFunc

;
```

28.16 enum SynapseMatrixConnectivity

< Flags defining differnet types of synaptic matrix connectivity

```
#include <synapseMatrixType.h>

enum SynapseMatrixConnectivity

    DENSE   = (1 <<0),
    BITMASK = (1 <<1),
    SPARSE  = (1 <<2),
;
```

28.17 enum SynapseMatrixType

```
#include <synapseMatrixType.h>

enum SynapseMatrixType

    DENSE_GLOBALG                   = static_cast<unsigned␣
↪int>(SynapseMatrixConnectivity::DENSE) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::GLOBAL),
    DENSE_GLOBALG_INDIVIDUAL_PSM    = static_cast<unsigned␣
↪int>(SynapseMatrixConnectivity::DENSE) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::GLOBAL) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::INDIVIDUAL_PSM),
    DENSE_INDIVIDUALG               = static_cast<unsigned␣
↪int>(SynapseMatrixConnectivity::DENSE) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::INDIVIDUAL) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::INDIVIDUAL_PSM),
    BITMASK_GLOBALG                 = static_cast<unsigned␣
↪int>(SynapseMatrixConnectivity::BITMASK) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::GLOBAL),
    BITMASK_GLOBALG_INDIVIDUAL_PSM = static_cast<unsigned␣
↪int>(SynapseMatrixConnectivity::BITMASK) | static_cast<unsigned␣
```

```
↪int>(SynapseMatrixWeight::GLOBAL) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::INDIVIDUAL_PSM),
    SPARSE_GLOBALG                 = static_cast<unsigned␣
↪int>(SynapseMatrixConnectivity::SPARSE) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::GLOBAL),
    SPARSE_GLOBALG_INDIVIDUAL_PSM  = static_cast<unsigned␣
↪int>(SynapseMatrixConnectivity::SPARSE) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::GLOBAL) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::INDIVIDUAL_PSM),
    SPARSE_INDIVIDUALG             =  static_cast<unsigned␣
↪int>(SynapseMatrixConnectivity::SPARSE) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::INDIVIDUAL) | static_cast<unsigned␣
↪int>(SynapseMatrixWeight::INDIVIDUAL_PSM),
;
```

28.18 enum SynapseMatrixWeight

```
#include <synapseMatrixType.h>

enum SynapseMatrixWeight

    GLOBAL         = (1 <<5),
    INDIVIDUAL     = (1 <<6),
    INDIVIDUAL_PSM = (1 <<7),
;
```

28.19 enum TimePrecision

28.19.1 Overview

Precision to use for variables which store time. *More...*

```
#include <modelSpec.h>

enum TimePrecision

    DEFAULT,
    FLOAT,
    DOUBLE,
;
```

28.19.2 Detailed Documentation

Precision to use for variables which store time.

Enum Values

DEFAULT

Time uses default model precision.

`FLOAT`

Time uses single precision - not suitable for long simulations.

`DOUBLE`

Time uses double precision - may reduce performance.

28.20 enum VarAccess

28.20.1 Overview

How is this variable accessed by model? *More...*

```
#include <models.h>

enum VarAccess

    READ_WRITE = 0,
    READ_ONLY,
;
```

28.20.2 Detailed Documentation

How is this variable accessed by model?

Enum Values

`READ_ONLY`

This variable is both read and written by the model.

28.21 enum VarLocation

< Flags defining which memory space variables should be allocated in

```
#include <variableMode.h>

enum VarLocation

    HOST                  = (1 <<0),
    DEVICE                = (1 <<1),
    ZERO_COPY             = (1 <<2),
    HOST_DEVICE           = HOST | DEVICE,
    HOST_DEVICE_ZERO_COPY = HOST | DEVICE | ZERO_COPY,
;
```

28.22 class CurrentSource

28.22.1 Overview

```
#include <currentSource.h>

class CurrentSource

public:
    // methods

    CurrentSource(const CurrentSource&);
    CurrentSource();
    void setVarLocation(const std::string& varName, VarLocation loc);
    void setExtraGlobalParamLocation(const std::string& paramName,␣
↪VarLocation loc);
    const std::string& getName() const;
    const CurrentSourceModels::Base* getCurrentSourceModel() const;
    const std::vector<double>& getParams() const;
    const std::vector<Models::VarInit>& getVarInitialisers() const;
    VarLocation getVarLocation(const std::string& varName) const;
    VarLocation getVarLocation(size_t index) const;
    VarLocation getExtraGlobalParamLocation(const std::string& paramName)␣
↪const;
    VarLocation getExtraGlobalParamLocation(size_t index) const;
;

// direct descendants

class CurrentSourceInternal;
```

28.22.2 Detailed Documentation

Methods

`void setVarLocation(const std::string& varName, VarLocation loc)`

Set location of current source state variable.

`void setExtraGlobalParamLocation(const std::string& paramName, VarLocation␣
↪loc)`

Set location of extra global parameter.

This is ignored for simulations on hardware with a single memory space and only applies to extra global parameters which are pointers.

`const CurrentSourceModels::Base* getCurrentSourceModel() const`

Gets the current source model used by this group.

`VarLocation getVarLocation(const std::string& varName) const`

Get variable location for current source model state variable.

`VarLocation getVarLocation(size_t index) const`

Get variable location for current source model state variable.

`VarLocation getExtraGlobalParamLocation(const std::string& paramName) const`

Get location of neuron model extra global parameter by name.

This is only used by extra global parameters which are pointers

`VarLocation getExtraGlobalParamLocation(size_t index) const`

Get location of neuron model extra global parameter by omdex.

This is only used by extra global parameters which are pointers

28.23 class CurrentSourceInternal

```
#include <currentSourceInternal.h>

class CurrentSourceInternal: public CurrentSource

public:
    // methods

    CurrentSourceInternal(
        const std::string& name,
        const CurrentSourceModels::Base* currentSourceModel,
        const std::vector<double>& params,
        const std::vector<Models::VarInit>& varInitialisers,
        VarLocation defaultVarLocation,
        VarLocation defaultExtraGlobalParamLocation
        );
;
```

28.23.1 Inherited Members

```
public:
    // methods

    CurrentSource(const CurrentSource&);
    CurrentSource();
    void setVarLocation(const std::string& varName, VarLocation loc);
    void setExtraGlobalParamLocation(const std::string& paramName,␣
↪VarLocation loc);
    const std::string& getName() const;
    const CurrentSourceModels::Base* getCurrentSourceModel() const;
    const std::vector<double>& getParams() const;
    const std::vector<Models::VarInit>& getVarInitialisers() const;
    VarLocation getVarLocation(const std::string& varName) const;
    VarLocation getVarLocation(size_t index) const;
    VarLocation getExtraGlobalParamLocation(const std::string& paramName)␣
↪const;
    VarLocation getExtraGlobalParamLocation(size_t index) const;
```

28.24 class ModelSpec

28.24.1 Overview

Object used for specifying a neuronal network model. *More...*

```
#include <modelSpec.h>

class ModelSpec

public:
    // typedefs

    typedef std::map<std::string, NeuronGroupInternal>::value_type␣
↪NeuronGroupValueType;
    typedef std::map<std::string, SynapseGroupInternal>::value_type␣
↪SynapseGroupValueType;

    // methods

    ModelSpec();
    ModelSpec(const ModelSpec&);
    ModelSpec& operator = (const ModelSpec&);
    ~ModelSpec();
    void setName(const std::string& name);
    void setPrecision(FloatType floattype);
    void setTimePrecision(TimePrecision timePrecision);
    void setDT(double dt);
    void setTiming(bool timingEnabled);
    void setSeed(unsigned int rngSeed);
    void setDefaultVarLocation(VarLocation loc);
    void setDefaultExtraGlobalParamLocation(VarLocation loc);
    void setDefaultSparseConnectivityLocation(VarLocation loc);
    void setMergePostsynapticModels(bool merge);
    const std::string& getName() const;
    const std::string& getPrecision() const;
    std::string getTimePrecision() const;
    double getDT() const;
    unsigned int getSeed() const;
    bool isTimingEnabled() const;
    unsigned int getNumLocalNeurons() const;
    unsigned int getNumRemoteNeurons() const;
    unsigned int getNumNeurons() const;
    NeuronGroup* findNeuronGroup(const std::string& name);

    template <typename NeuronModel>
    NeuronGroup* addNeuronPopulation(
        const std::string& name,
        unsigned int size,
        const NeuronModel* model,
        const typename NeuronModel::ParamValues& paramValues,
        const typename NeuronModel::VarValues& varInitialisers,
        int hostID = 0
```

```
        );

    template <typename NeuronModel>
    NeuronGroup* addNeuronPopulation(
        const std::string& name,
        unsigned int size,
        const typename NeuronModel::ParamValues& paramValues,
        const typename NeuronModel::VarValues& varInitialisers,
        int hostID = 0
        );

    SynapseGroup* findSynapseGroup(const std::string& name);

    template <typename WeightUpdateModel, typename PostsynapticModel>
    SynapseGroup* addSynapsePopulation(
        const std::string& name,
        SynapseMatrixType mtype,
        unsigned int delaySteps,
        const std::string& src,
        const std::string& trg,
        const WeightUpdateModel* wum,
        const typename WeightUpdateModel::ParamValues& weightParamValues,
        const typename WeightUpdateModel::VarValues& weightVarInitialisers,
        const typename WeightUpdateModel::PreVarValues& weightPreVarInitialisers,
↪
        const typename WeightUpdateModel::PostVarValues&␣
↪weightPostVarInitialisers,
        const PostsynapticModel* psm,
        const typename PostsynapticModel::ParamValues& postsynapticParamValues,
↪
        const typename PostsynapticModel::VarValues& postsynapticVarInitialisers,
↪
        const InitSparseConnectivitySnippet::Init& connectivityInitialiser =␣
↪uninitialisedConnectivity()
        );

    template <typename WeightUpdateModel, typename PostsynapticModel>
    SynapseGroup* addSynapsePopulation(
        const std::string& name,
        SynapseMatrixType mtype,
        unsigned int delaySteps,
        const std::string& src,
        const std::string& trg,
        const typename WeightUpdateModel::ParamValues& weightParamValues,
        const typename WeightUpdateModel::VarValues& weightVarInitialisers,
        const typename PostsynapticModel::ParamValues& postsynapticParamValues,
↪
        const typename PostsynapticModel::VarValues& postsynapticVarInitialisers,
↪
        const InitSparseConnectivitySnippet::Init& connectivityInitialiser =␣
↪uninitialisedConnectivity()
        );

    template <typename WeightUpdateModel, typename PostsynapticModel>
```

```
    SynapseGroup* addSynapsePopulation(
        const std::string& name,
        SynapseMatrixType mtype,
        unsigned int delaySteps,
        const std::string& src,
        const std::string& trg,
        const typename WeightUpdateModel::ParamValues& weightParamValues,
        const typename WeightUpdateModel::VarValues& weightVarInitialisers,
        const typename WeightUpdateModel::PreVarValues& weightPreVarInitialisers,
↪
        const typename WeightUpdateModel::PostVarValues&␣
↪weightPostVarInitialisers,
        const typename PostsynapticModel::ParamValues& postsynapticParamValues,
↪
        const typename PostsynapticModel::VarValues& postsynapticVarInitialisers,
↪
        const InitSparseConnectivitySnippet::Init& connectivityInitialiser =␣
↪uninitialisedConnectivity()
        );

    CurrentSource* findCurrentSource(const std::string& name);

    template <typename CurrentSourceModel>
    CurrentSource* addCurrentSource(
        const std::string& currentSourceName,
        const CurrentSourceModel* model,
        const std::string& targetNeuronGroupName,
        const typename CurrentSourceModel::ParamValues& paramValues,
        const typename CurrentSourceModel::VarValues& varInitialisers
        );

    template <typename CurrentSourceModel>
    CurrentSource* addCurrentSource(
        const std::string& currentSourceName,
        const std::string& targetNeuronGroupName,
        const typename CurrentSourceModel::ParamValues& paramValues,
        const typename CurrentSourceModel::VarValues& varInitialisers
        );
;

// direct descendants

class ModelSpecInternal;
```

28.24.2 Detailed Documentation

Object used for specifying a neuronal network model.

Methods

`void setName(const std::string& name)`

Method to set the neuronal network model name.

`void setPrecision(FloatType floattype)`

Set numerical precision for floating point.

This function sets the numerical precision of floating type variables. By default, it is GENN_GENN_FLOAT.

`void setTimePrecision(TimePrecision timePrecision)`

Set numerical precision for time.

`void setDT(double dt)`

Set the integration step size of the model.

`void setTiming(bool timingEnabled)`

Set whether timers and timing commands are to be included.

`void setSeed(unsigned int rngSeed)`

Set the random seed (disables automatic seeding if argument not 0).

`void setDefaultVarLocation(VarLocation loc)`

What is the default location for model state variables?

Historically, everything was allocated on both the host AND device

`void setDefaultExtraGlobalParamLocation(VarLocation loc)`

What is the default location for model extra global parameters?

Historically, this was just left up to the user to handle

`void setDefaultSparseConnectivityLocation(VarLocation loc)`

What is the default location for sparse synaptic connectivity?

Historically, everything was allocated on both the host AND device

`void setMergePostsynapticModels(bool merge)`

Should compatible postsynaptic models and dendritic delay buffers be merged?

This can significantly reduce the cost of updating neuron population but means that per-synapse group inSyn arrays can not be retrieved

`const std::string& getName() const`

Gets the name of the neuronal network model.

`const std::string& getPrecision() const`

Gets the floating point numerical precision.

`std::string getTimePrecision() const`

Gets the floating point numerical precision used to represent time.

`double getDT() const`

Gets the model integration step size.

`unsigned int getSeed() const`

Get the random seed.

`bool isTimingEnabled() const`

Are timers and timing commands enabled.

```
unsigned int getNumLocalNeurons() const
```

How many neurons are simulated locally in this model.

```
unsigned int getNumRemoteNeurons() const
```

How many neurons are simulated remotely in this model.

```
unsigned int getNumNeurons() const
```

How many neurons make up the entire model.

```
NeuronGroup* findNeuronGroup(const std::string& name)
```

Find a neuron group by name.

```
template <typename NeuronModel>
NeuronGroup* addNeuronPopulation(
    const std::string& name,
    unsigned int size,
    const NeuronModel* model,
    const typename NeuronModel::ParamValues& paramValues,
    const typename NeuronModel::VarValues& varInitialisers,
    int hostID = 0
    )
```

Adds a new neuron group to the model using a neuron model managed by the user.

Parameters:

NeuronModel	type of neuron model (derived from *NeuronModels::Base*).
name	string containing unique name of neuron population.
size	integer specifying how many neurons are in the population.
model	neuron model to use for neuron group.
paramValues	parameters for model wrapped in NeuronModel::ParamValues object.
varInitialisers	state variable initialiser snippets and parameters wrapped in NeuronModel::VarValues object.
hostID	if using MPI, the ID of the node to simulate this population on.

Returns:

pointer to newly created *NeuronGroup*

```
template <typename NeuronModel>
NeuronGroup* addNeuronPopulation(
    const std::string& name,
    unsigned int size,
    const typename NeuronModel::ParamValues& paramValues,
    const typename NeuronModel::VarValues& varInitialisers,
    int hostID = 0
    )
```

Adds a new neuron group to the model using a singleton neuron model created using standard DECLARE_MODEL and IMPLEMENT_MODEL macros.

Parameters:

NeuronModel	type of neuron model (derived from *NeuronModels::Base*).
name	string containing unique name of neuron population.
size	integer specifying how many neurons are in the population.
paramValues	parameters for model wrapped in NeuronModel::ParamValues object.
varInitialisers	state variable initialiser snippets and parameters wrapped in NeuronModel::VarValues object.
hostID	if using MPI, the ID of the node to simulate this population on.

Returns:

pointer to newly created *NeuronGroup*

```
SynapseGroup* findSynapseGroup(const std::string& name)
```

Find a synapse group by name.

```
template <typename WeightUpdateModel, typename PostsynapticModel>
SynapseGroup* addSynapsePopulation(
    const std::string& name,
    SynapseMatrixType mtype,
    unsigned int delaySteps,
    const std::string& src,
    const std::string& trg,
    const WeightUpdateModel* wum,
    const typename WeightUpdateModel::ParamValues& weightParamValues,
    const typename WeightUpdateModel::VarValues& weightVarInitialisers,
    const typename WeightUpdateModel::PreVarValues& weightPreVarInitialisers,
    const typename WeightUpdateModel::PostVarValues& weightPostVarInitialisers,
↪
    const PostsynapticModel* psm,
    const typename PostsynapticModel::ParamValues& postsynapticParamValues,
    const typename PostsynapticModel::VarValues& postsynapticVarInitialisers,
    const InitSparseConnectivitySnippet::Init& connectivityInitialiser =␣
↪uninitialisedConnectivity()
    )
```

Adds a synapse population to the model using weight update and postsynaptic models managed by the user.

Parameters:

WeightUpdateModel	type of weight update model (derived from *WeightUpdateModels::Base*).
PostsynapticModel	type of postsynaptic model (derived from *PostsynapticModels::Base*).
name	string containing unique name of neuron population.
mtype	how the synaptic matrix associated with this synapse population should be represented.
delaySteps	integer specifying number of timesteps delay this synaptic connection should incur (or NO_DELAY for none)
src	string specifying name of presynaptic (source) population
trg	string specifying name of postsynaptic (target) population
wum	weight update model to use for synapse group.
weightParamValues	parameters for weight update model wrapped in WeightUpdateModel::ParamValues object.
weightVarInitialisers	weight update model state variable initialiser snippets and parameters wrapped in WeightUpdateModel::VarValues object.
weightPreVarInitialisers	weight update model presynaptic state variable initialiser snippets and parameters wrapped in WeightUpdateModel::VarValues object.
weightPostVarInitialisers	weight update model postsynaptic state variable initialiser snippets and parameters wrapped in WeightUpdateModel::VarValues object.
psm	postsynaptic model to use for synapse group.
postsynapticParamValues	parameters for postsynaptic model wrapped in PostsynapticModel::ParamValues object.
postsynapticVarInitialisers	postsynaptic model state variable initialiser snippets and parameters wrapped in NeuronModel::VarValues object.
connectivityInitialiser	sparse connectivity initialisation snippet used to initialise connectivity for SynapseMatrixConnectivity::SPARSE or SynapseMatrixConnectivity::BITMASK. Typically wrapped with it's parameters using `initConnectivity` function

Returns:

pointer to newly created *SynapseGroup*

```
template <typename WeightUpdateModel, typename PostsynapticModel>
SynapseGroup* addSynapsePopulation(
    const std::string& name,
    SynapseMatrixType mtype,
    unsigned int delaySteps,
    const std::string& src,
    const std::string& trg,
    const typename WeightUpdateModel::ParamValues& weightParamValues,
    const typename WeightUpdateModel::VarValues& weightVarInitialisers,
    const typename PostsynapticModel::ParamValues& postsynapticParamValues,
    const typename PostsynapticModel::VarValues& postsynapticVarInitialisers,
    const InitSparseConnectivitySnippet::Init& connectivityInitialiser =␣
↪uninitialisedConnectivity()
    )
```

Adds a synapse population to the model using singleton weight update and postsynaptic models created using standard DECLARE_MODEL and IMPLEMENT_MODEL macros.

Parameters:

WeightUpdateModel	type of weight update model (derived from *WeightUpdateModels::Base*).
PostsynapticModel	type of postsynaptic model (derived from *PostsynapticModels::Base*).
name	string containing unique name of neuron population.
mtype	how the synaptic matrix associated with this synapse population should be represented.
delaySteps	integer specifying number of timesteps delay this synaptic connection should incur (or NO_DELAY for none)
src	string specifying name of presynaptic (source) population
trg	string specifying name of postsynaptic (target) population
weightParamValues	parameters for weight update model wrapped in WeightUpdateModel::ParamValues object.
weightVarInitialisers	weight update model state variable initialiser snippets and parameters wrapped in WeightUpdateModel::VarValues object.
postsynapticParamValues	parameters for postsynaptic model wrapped in PostsynapticModel::ParamValues object.
postsynapticVarInitialisers	postsynaptic model state variable initialiser snippets and parameters wrapped in NeuronModel::VarValues object.
connectivityInitialiser	sparse connectivity initialisation snippet used to initialise connectivity for SynapseMatrixConnectivity::SPARSE or SynapseMatrixConnectivity::BITMASK. Typically wrapped with it's parameters using `initConnectivity` function

Returns:

pointer to newly created *SynapseGroup*

```
template <typename WeightUpdateModel, typename PostsynapticModel>
SynapseGroup* addSynapsePopulation(
    const std::string& name,
    SynapseMatrixType mtype,
    unsigned int delaySteps,
    const std::string& src,
    const std::string& trg,
    const typename WeightUpdateModel::ParamValues& weightParamValues,
    const typename WeightUpdateModel::VarValues& weightVarInitialisers,
    const typename WeightUpdateModel::PreVarValues& weightPreVarInitialisers,
    const typename WeightUpdateModel::PostVarValues& weightPostVarInitialisers,
↪
    const typename PostsynapticModel::ParamValues& postsynapticParamValues,
    const typename PostsynapticModel::VarValues& postsynapticVarInitialisers,
    const InitSparseConnectivitySnippet::Init& connectivityInitialiser =␣
↪uninitialisedConnectivity()
    )
```

Adds a synapse population to the model using singleton weight update and postsynaptic models created using standard DECLARE_MODEL and IMPLEMENT_MODEL macros.

Parameters:

WeightUpdateModel	type of weight update model (derived from *WeightUpdateModels::Base*).
PostsynapticModel	type of postsynaptic model (derived from *PostsynapticModels::Base*).
name	string containing unique name of neuron population.
mtype	how the synaptic matrix associated with this synapse population should be represented.
delaySteps	integer specifying number of timesteps delay this synaptic connection should incur (or NO_DELAY for none)
src	string specifying name of presynaptic (source) population
trg	string specifying name of postsynaptic (target) population
weightParamValues	parameters for weight update model wrapped in WeightUpdateModel::ParamValues object.
weightVarInitialisers	weight update model per-synapse state variable initialiser snippets and parameters wrapped in WeightUpdateModel::VarValues object.
weightPreVarInitialisers	weight update model presynaptic state variable initialiser snippets and parameters wrapped in WeightUpdateModel::VarValues object.
weightPostVarInitialisers	weight update model postsynaptic state variable initialiser snippets and parameters wrapped in WeightUpdateModel::VarValues object.
postsynapticParamValues	parameters for postsynaptic model wrapped in PostsynapticModel::ParamValues object.
postsynapticVarInitialisers	postsynaptic model state variable initialiser snippets and parameters wrapped in NeuronModel::VarValues object.
connectivityInitialiser	sparse connectivity initialisation snippet used to initialise connectivity for SynapseMatrixConnectivity::SPARSE or SynapseMatrixConnectivity::BITMASK. Typically wrapped with it's parameters using `initConnectivity` function

Returns:

pointer to newly created *SynapseGroup*

```
CurrentSource* findCurrentSource(const std::string& name)
```

Find a current source by name.

This function attempts to find an existing current source.

```
template <typename CurrentSourceModel>
CurrentSource* addCurrentSource(
    const std::string& currentSourceName,
    const CurrentSourceModel* model,
    const std::string& targetNeuronGroupName,
    const typename CurrentSourceModel::ParamValues& paramValues,
    const typename CurrentSourceModel::VarValues& varInitialisers
    )
```

Adds a new current source to the model using a current source model managed by the user.

Parameters:

CurrentSourceModel	type of current source model (derived from *CurrentSourceModels::Base*).
currentSourceName	string containing unique name of current source.
model	current source model to use for current source.
targetNeuronGroupName	string name of the target neuron group
paramValues	parameters for model wrapped in CurrentSourceModel::ParamValues object.
varInitialisers	state variable initialiser snippets and parameters wrapped in CurrentSource::VarValues object.

Returns:

pointer to newly created *CurrentSource*

```
template <typename CurrentSourceModel>
CurrentSource* addCurrentSource(
    const std::string& currentSourceName,
    const std::string& targetNeuronGroupName,
    const typename CurrentSourceModel::ParamValues& paramValues,
    const typename CurrentSourceModel::VarValues& varInitialisers
    )
```

Adds a new current source to the model using a singleton current source model created using standard DECLARE_MODEL and IMPLEMENT_MODEL macros.

Parameters:

CurrentSourceModel	type of neuron model (derived from CurrentSourceModel::Base).
currentSourceName	string containing unique name of current source.
targetNeuronGroupName	string name of the target neuron group
paramValues	parameters for model wrapped in CurrentSourceModel::ParamValues object.
varInitialisers	state variable initialiser snippets and parameters wrapped in CurrentSourceModel::VarValues object.

Returns:

pointer to newly created *CurrentSource*

28.25 class ModelSpecInternal

```
#include <modelSpecInternal.h>

class ModelSpecInternal: public ModelSpec

;
```

28.25.1 Inherited Members

```
public:
    // typedefs

    typedef std::map<std::string, NeuronGroupInternal>::value_type␣
↪NeuronGroupValueType;
    typedef std::map<std::string, SynapseGroupInternal>::value_type␣
↪SynapseGroupValueType;

    // methods

    ModelSpec();
    ModelSpec(const ModelSpec&);
    ModelSpec& operator = (const ModelSpec&);
    ~ModelSpec();
    void setName(const std::string& name);
    void setPrecision(FloatType floattype);
    void setTimePrecision(TimePrecision timePrecision);
    void setDT(double dt);
    void setTiming(bool timingEnabled);
    void setSeed(unsigned int rngSeed);
    void setDefaultVarLocation(VarLocation loc);
    void setDefaultExtraGlobalParamLocation(VarLocation loc);
    void setDefaultSparseConnectivityLocation(VarLocation loc);
    void setMergePostsynapticModels(bool merge);
    const std::string& getName() const;
    const std::string& getPrecision() const;
    std::string getTimePrecision() const;
    double getDT() const;
    unsigned int getSeed() const;
    bool isTimingEnabled() const;
    unsigned int getNumLocalNeurons() const;
    unsigned int getNumRemoteNeurons() const;
    unsigned int getNumNeurons() const;
    NeuronGroup* findNeuronGroup(const std::string& name);

    template <typename NeuronModel>
    NeuronGroup* addNeuronPopulation(
        const std::string& name,
        unsigned int size,
        const NeuronModel* model,
        const typename NeuronModel::ParamValues& paramValues,
        const typename NeuronModel::VarValues& varInitialisers,
        int hostID = 0
        );

    template <typename NeuronModel>
    NeuronGroup* addNeuronPopulation(
        const std::string& name,
        unsigned int size,
        const typename NeuronModel::ParamValues& paramValues,
        const typename NeuronModel::VarValues& varInitialisers,
        int hostID = 0
```

```
        );

    SynapseGroup* findSynapseGroup(const std::string& name);

    template <typename WeightUpdateModel, typename PostsynapticModel>
    SynapseGroup* addSynapsePopulation(
        const std::string& name,
        SynapseMatrixType mtype,
        unsigned int delaySteps,
        const std::string& src,
        const std::string& trg,
        const WeightUpdateModel* wum,
        const typename WeightUpdateModel::ParamValues& weightParamValues,
        const typename WeightUpdateModel::VarValues& weightVarInitialisers,
        const typename WeightUpdateModel::PreVarValues& weightPreVarInitialisers,
↪
        const typename WeightUpdateModel::PostVarValues&␣
↪weightPostVarInitialisers,
        const PostsynapticModel* psm,
        const typename PostsynapticModel::ParamValues& postsynapticParamValues,
↪
        const typename PostsynapticModel::VarValues& postsynapticVarInitialisers,
↪
        const InitSparseConnectivitySnippet::Init& connectivityInitialiser =␣
↪uninitialisedConnectivity()
        );

    template <typename WeightUpdateModel, typename PostsynapticModel>
    SynapseGroup* addSynapsePopulation(
        const std::string& name,
        SynapseMatrixType mtype,
        unsigned int delaySteps,
        const std::string& src,
        const std::string& trg,
        const typename WeightUpdateModel::ParamValues& weightParamValues,
        const typename WeightUpdateModel::VarValues& weightVarInitialisers,
        const typename PostsynapticModel::ParamValues& postsynapticParamValues,
↪
        const typename PostsynapticModel::VarValues& postsynapticVarInitialisers,
↪
        const InitSparseConnectivitySnippet::Init& connectivityInitialiser =␣
↪uninitialisedConnectivity()
        );

    template <typename WeightUpdateModel, typename PostsynapticModel>
    SynapseGroup* addSynapsePopulation(
        const std::string& name,
        SynapseMatrixType mtype,
        unsigned int delaySteps,
        const std::string& src,
        const std::string& trg,
        const typename WeightUpdateModel::ParamValues& weightParamValues,
        const typename WeightUpdateModel::VarValues& weightVarInitialisers,
        const typename WeightUpdateModel::PreVarValues& weightPreVarInitialisers,
```

```
↪
        const typename WeightUpdateModel::PostVarValues&␣
↪weightPostVarInitialisers,
        const typename PostsynapticModel::ParamValues& postsynapticParamValues,
↪
        const typename PostsynapticModel::VarValues& postsynapticVarInitialisers,
↪
        const InitSparseConnectivitySnippet::Init& connectivityInitialiser =␣
↪uninitialisedConnectivity()
        );

    CurrentSource* findCurrentSource(const std::string& name);

    template <typename CurrentSourceModel>
    CurrentSource* addCurrentSource(
        const std::string& currentSourceName,
        const CurrentSourceModel* model,
        const std::string& targetNeuronGroupName,
        const typename CurrentSourceModel::ParamValues& paramValues,
        const typename CurrentSourceModel::VarValues& varInitialisers
        );

    template <typename CurrentSourceModel>
    CurrentSource* addCurrentSource(
        const std::string& currentSourceName,
        const std::string& targetNeuronGroupName,
        const typename CurrentSourceModel::ParamValues& paramValues,
        const typename CurrentSourceModel::VarValues& varInitialisers
        );
```

28.26 class NeuronGroup

28.26.1 Overview

```
#include <neuronGroup.h>

class NeuronGroup

public:
    // methods

    NeuronGroup(const NeuronGroup&);
    NeuronGroup();
    void setSpikeLocation(VarLocation loc);
    void setSpikeEventLocation(VarLocation loc);
    void setSpikeTimeLocation(VarLocation loc);
    void setVarLocation(const std::string& varName, VarLocation loc);
    void setExtraGlobalParamLocation(const std::string& paramName,␣
↪VarLocation loc);
    const std::string& getName() const;
    unsigned int getNumNeurons() const;
    const NeuronModels::Base* getNeuronModel() const;
```

```
    const std::vector<double>& getParams() const;
    const std::vector<Models::VarInit>& getVarInitialisers() const;
    int getClusterHostID() const;
    bool isSpikeTimeRequired() const;
    bool isTrueSpikeRequired() const;
    bool isSpikeEventRequired() const;
    unsigned int getNumDelaySlots() const;
    bool isDelayRequired() const;
    bool isZeroCopyEnabled() const;
    VarLocation getSpikeLocation() const;
    VarLocation getSpikeEventLocation() const;
    VarLocation getSpikeTimeLocation() const;
    VarLocation getVarLocation(const std::string& varName) const;
    VarLocation getVarLocation(size_t index) const;
    VarLocation getExtraGlobalParamLocation(const std::string& paramName)␣
↪const;
    VarLocation getExtraGlobalParamLocation(size_t index) const;
    bool isSimRNGRequired() const;
    bool isInitRNGRequired() const;
    bool hasOutputToHost(int targetHostID) const;
;

// direct descendants

class NeuronGroupInternal;
```

28.26.2 Detailed Documentation

Methods

`void setSpikeLocation(VarLocation loc)`

Set location of this neuron group's output spikes.

This is ignored for simulations on hardware with a single memory space

`void setSpikeEventLocation(VarLocation loc)`

Set location of this neuron group's output spike events.

This is ignored for simulations on hardware with a single memory space

`void setSpikeTimeLocation(VarLocation loc)`

Set location of this neuron group's output spike times.

This is ignored for simulations on hardware with a single memory space

`void setVarLocation(const std::string& varName, VarLocation loc)`

Set variable location of neuron model state variable.

This is ignored for simulations on hardware with a single memory space

`void setExtraGlobalParamLocation(const std::string& paramName, VarLocation␣↪loc)`

Set location of neuron model extra global parameter.

This is ignored for simulations on hardware with a single memory space and only applies to extra global parameters which are pointers.

`unsigned int getNumNeurons() const`

Gets number of neurons in group.

`const NeuronModels::Base* getNeuronModel() const`

Gets the neuron model used by this group.

`VarLocation getSpikeLocation() const`

Get location of this neuron group's output spikes.

`VarLocation getSpikeEventLocation() const`

Get location of this neuron group's output spike events.

`VarLocation getSpikeTimeLocation() const`

Get location of this neuron group's output spike times.

`VarLocation getVarLocation(const std::string& varName) const`

Get location of neuron model state variable by name.

`VarLocation getVarLocation(size_t index) const`

Get location of neuron model state variable by index.

`VarLocation getExtraGlobalParamLocation(const std::string& paramName) const`

Get location of neuron model extra global parameter by name.

This is only used by extra global parameters which are pointers

`VarLocation getExtraGlobalParamLocation(size_t index) const`

Get location of neuron model extra global parameter by omdex.

This is only used by extra global parameters which are pointers

`bool isSimRNGRequired() const`

Does this neuron group require an RNG to simulate?

`bool isInitRNGRequired() const`

Does this neuron group require an RNG for it's init code?

`bool hasOutputToHost(int targetHostID) const`

Does this neuron group have outgoing connections specified host id?

28.27 class NeuronGroupInternal

```
#include <neuronGroupInternal.h>

class NeuronGroupInternal: public NeuronGroup

public:
    // methods

    NeuronGroupInternal(
```

```
        const std::string& name,
        int numNeurons,
        const NeuronModels::Base* neuronModel,
        const std::vector<double>& params,
        const std::vector<Models::VarInit>& varInitialisers,
        VarLocation defaultVarLocation,
        VarLocation defaultExtraGlobalParamLocation,
        int hostID
        );
;
```

28.27.1 Inherited Members

```
public:
    // methods

    NeuronGroup(const NeuronGroup&);
    NeuronGroup();
    void setSpikeLocation(VarLocation loc);
    void setSpikeEventLocation(VarLocation loc);
    void setSpikeTimeLocation(VarLocation loc);
    void setVarLocation(const std::string& varName, VarLocation loc);
    void setExtraGlobalParamLocation(const std::string& paramName,␣
↪VarLocation loc);
    const std::string& getName() const;
    unsigned int getNumNeurons() const;
    const NeuronModels::Base* getNeuronModel() const;
    const std::vector<double>& getParams() const;
    const std::vector<Models::VarInit>& getVarInitialisers() const;
    int getClusterHostID() const;
    bool isSpikeTimeRequired() const;
    bool isTrueSpikeRequired() const;
    bool isSpikeEventRequired() const;
    unsigned int getNumDelaySlots() const;
    bool isDelayRequired() const;
    bool isZeroCopyEnabled() const;
    VarLocation getSpikeLocation() const;
    VarLocation getSpikeEventLocation() const;
    VarLocation getSpikeTimeLocation() const;
    VarLocation getVarLocation(const std::string& varName) const;
    VarLocation getVarLocation(size_t index) const;
    VarLocation getExtraGlobalParamLocation(const std::string& paramName)␣
↪const;
    VarLocation getExtraGlobalParamLocation(size_t index) const;
    bool isSimRNGRequired() const;
    bool isInitRNGRequired() const;
    bool hasOutputToHost(int targetHostID) const;
```

28.28 class SynapseGroup

28.28.1 enum SynapseGroup::SpanType

```
#include <synapseGroup.h>

enum SpanType

    POSTSYNAPTIC,
    PRESYNAPTIC,
;
```

28.28.2 Overview

```
#include <synapseGroup.h>

class SynapseGroup

public:
    // enums

    enum SpanType;

    // methods

    SynapseGroup(const SynapseGroup&);
    SynapseGroup();
    void setWUVarLocation(const std::string& varName, VarLocation loc);
    void setWUPreVarLocation(const std::string& varName, VarLocation loc);
    void setWUPostVarLocation(const std::string& varName, VarLocation loc);
    void setWUExtraGlobalParamLocation(const std::string& paramName,␣
↪VarLocation loc);
    void setPSVarLocation(const std::string& varName, VarLocation loc);
    void setPSExtraGlobalParamLocation(const std::string& paramName,␣
↪VarLocation loc);
    void setSparseConnectivityExtraGlobalParamLocation(const std::string&␣
↪paramName, VarLocation loc);
    void setInSynVarLocation(VarLocation loc);
    void setSparseConnectivityLocation(VarLocation loc);
    void setDendriticDelayLocation(VarLocation loc);
    void setMaxConnections(unsigned int maxConnections);
    void setMaxSourceConnections(unsigned int maxPostConnections);
    void setMaxDendriticDelayTimesteps(unsigned int maxDendriticDelay);
    void setSpanType(SpanType spanType);
    void setNumThreadsPerSpike(unsigned int numThreadsPerSpike);
    void setBackPropDelaySteps(unsigned int timesteps);
    const std::string& getName() const;
    SpanType getSpanType() const;
    unsigned int getNumThreadsPerSpike() const;
    unsigned int getDelaySteps() const;
    unsigned int getBackPropDelaySteps() const;
    unsigned int getMaxConnections() const;
```

```
    unsigned int getMaxSourceConnections() const;
    unsigned int getMaxDendriticDelayTimesteps() const;
    SynapseMatrixType getMatrixType() const;
    VarLocation getInSynLocation() const;
    VarLocation getSparseConnectivityLocation() const;
    VarLocation getDendriticDelayLocation() const;
    int getClusterHostID() const;
    bool isTrueSpikeRequired() const;
    bool isSpikeEventRequired() const;
    const WeightUpdateModels::Base* getWUModel() const;
    const std::vector<double>& getWUParams() const;
    const std::vector<Models::VarInit>& getWUVarInitialisers() const;
    const std::vector<Models::VarInit>& getWUPreVarInitialisers() const;
    const std::vector<Models::VarInit>& getWUPostVarInitialisers() const;
    const std::vector<double> getWUConstInitVals() const;
    const PostsynapticModels::Base* getPSModel() const;
    const std::vector<double>& getPSParams() const;
    const std::vector<Models::VarInit>& getPSVarInitialisers() const;
    const std::vector<double> getPSConstInitVals() const;
    const InitSparseConnectivitySnippet::Init& getConnectivityInitialiser()␣
↪const;
    bool isZeroCopyEnabled() const;
    VarLocation getWUVarLocation(const std::string& var) const;
    VarLocation getWUVarLocation(size_t index) const;
    VarLocation getWUPreVarLocation(const std::string& var) const;
    VarLocation getWUPreVarLocation(size_t index) const;
    VarLocation getWUPostVarLocation(const std::string& var) const;
    VarLocation getWUPostVarLocation(size_t index) const;
    VarLocation getWUExtraGlobalParamLocation(const std::string& paramName)␣
↪const;
    VarLocation getWUExtraGlobalParamLocation(size_t index) const;
    VarLocation getPSVarLocation(const std::string& var) const;
    VarLocation getPSVarLocation(size_t index) const;
    VarLocation getPSExtraGlobalParamLocation(const std::string& paramName)␣
↪const;
    VarLocation getPSExtraGlobalParamLocation(size_t index) const;
    VarLocation getSparseConnectivityExtraGlobalParamLocation(const␣
↪std::string& paramName) const;
    VarLocation getSparseConnectivityExtraGlobalParamLocation(size_t index)␣
↪const;
    bool isDendriticDelayRequired() const;
    bool isPSInitRNGRequired() const;
    bool isWUInitRNGRequired() const;
    bool isWUVarInitRequired() const;
    bool isSparseConnectivityInitRequired() const;
;

// direct descendants

class SynapseGroupInternal;
```

28.28.3 Detailed Documentation

Methods

`void setWUVarLocation(const std::string& varName, VarLocation loc)`

Set location of weight update model state variable.

This is ignored for simulations on hardware with a single memory space

`void setWUPreVarLocation(const std::string& varName, VarLocation loc)`

Set location of weight update model presynaptic state variable.

This is ignored for simulations on hardware with a single memory space

`void setWUPostVarLocation(const std::string& varName, VarLocation loc)`

Set location of weight update model postsynaptic state variable.

This is ignored for simulations on hardware with a single memory space

```
void setWUExtraGlobalParamLocation(const std::string& paramName, VarLocation␣
↪loc)
```

Set location of weight update model extra global parameter.

This is ignored for simulations on hardware with a single memory space and only applies to extra global parameters which are pointers.

`void setPSVarLocation(const std::string& varName, VarLocation loc)`

Set location of postsynaptic model state variable.

This is ignored for simulations on hardware with a single memory space

```
void setPSExtraGlobalParamLocation(const std::string& paramName, VarLocation␣
↪loc)
```

Set location of postsynaptic model extra global parameter.

This is ignored for simulations on hardware with a single memory space and only applies to extra global parameters which are pointers.

```
void setSparseConnectivityExtraGlobalParamLocation(
    const std::string& paramName,
    VarLocation loc
    )
```

Set location of sparse connectivity initialiser extra global parameter.

This is ignored for simulations on hardware with a single memory space and only applies to extra global parameters which are pointers.

`void setInSynVarLocation(VarLocation loc)`

Set location of variables used to combine input from this synapse group.

This is ignored for simulations on hardware with a single memory space

`void setSparseConnectivityLocation(VarLocation loc)`

Set variable mode used for sparse connectivity.

This is ignored for simulations on hardware with a single memory space

`void setDendriticDelayLocation(VarLocation loc)`

Set variable mode used for this synapse group's dendritic delay buffers.

`void setMaxConnections(unsigned int maxConnections)`

Sets the maximum number of target neurons any source neurons can connect to.

Use with synaptic matrix types with SynapseMatrixConnectivity::SPARSE to optimise CUDA implementation

`void setMaxSourceConnections(unsigned int maxPostConnections)`

Sets the maximum number of source neurons any target neuron can connect to.

Use with synaptic matrix types with SynapseMatrixConnectivity::SPARSE and postsynaptic learning to optimise CUDA implementation

`void setMaxDendriticDelayTimesteps(unsigned int maxDendriticDelay)`

Sets the maximum dendritic delay for synapses in this synapse group.

`void setSpanType(SpanType spanType)`

Set how CUDA implementation is parallelised.

with a thread per target neuron (default) or a thread per source spike

`void setNumThreadsPerSpike(unsigned int numThreadsPerSpike)`

Set how many threads CUDA implementation uses to process each spike when span type is PRESYNAPTIC.

`void setBackPropDelaySteps(unsigned int timesteps)`

Sets the number of delay steps used to delay postsynaptic spikes travelling back along dendrites to synapses.

`VarLocation getInSynLocation() const`

Get variable mode used for variables used to combine input from this synapse group.

`VarLocation getSparseConnectivityLocation() const`

Get variable mode used for sparse connectivity.

`VarLocation getDendriticDelayLocation() const`

Get variable mode used for this synapse group's dendritic delay buffers.

`bool isTrueSpikeRequired() const`

Does synapse group need to handle 'true' spikes.

`bool isSpikeEventRequired() const`

Does synapse group need to handle spike-like events.

`VarLocation getWUVarLocation(const std::string& var) const`

Get location of weight update model per-synapse state variable by name.

`VarLocation getWUVarLocation(size_t index) const`

Get location of weight update model per-synapse state variable by index.

`VarLocation getWUPreVarLocation(const std::string& var) const`

Get location of weight update model presynaptic state variable by name.

`VarLocation getWUPreVarLocation(size_t index) const`

Get location of weight update model presynaptic state variable by index.

`VarLocation getWUPostVarLocation(const std::string& var) const`

Get location of weight update model postsynaptic state variable by name.

`VarLocation getWUPostVarLocation(size_t index) const`

Get location of weight update model postsynaptic state variable by index.

`VarLocation getWUExtraGlobalParamLocation(const std::string& paramName) const`

Get location of weight update model extra global parameter by name.

This is only used by extra global parameters which are pointers

`VarLocation getWUExtraGlobalParamLocation(size_t index) const`

Get location of weight update model extra global parameter by index.

This is only used by extra global parameters which are pointers

`VarLocation getPSVarLocation(const std::string& var) const`

Get location of postsynaptic model state variable.

`VarLocation getPSVarLocation(size_t index) const`

Get location of postsynaptic model state variable.

`VarLocation getPSExtraGlobalParamLocation(const std::string& paramName) const`

Get location of postsynaptic model extra global parameter by name.

This is only used by extra global parameters which are pointers

`VarLocation getPSExtraGlobalParamLocation(size_t index) const`

Get location of postsynaptic model extra global parameter by index.

This is only used by extra global parameters which are pointers

`VarLocation getSparseConnectivityExtraGlobalParamLocation(const std::string& paramName) const`

Get location of sparse connectivity initialiser extra global parameter by name.

This is only used by extra global parameters which are pointers

`VarLocation getSparseConnectivityExtraGlobalParamLocation(size_t index) const`

Get location of sparse connectivity initialiser extra global parameter by index.

This is only used by extra global parameters which are pointers

`bool isDendriticDelayRequired() const`

Does this synapse group require dendritic delay?

`bool isPSInitRNGRequired() const`

Does this synapse group require an RNG for it's postsynaptic init code?

`bool isWUInitRNGRequired() const`

Does this synapse group require an RNG for it's weight update init code?

`bool isWUVarInitRequired() const`

Is var init code required for any variables in this synapse group's weight update model?

`bool isSparseConnectivityInitRequired() const`

Is sparse connectivity initialisation code required for this synapse group?

28.29 class SynapseGroupInternal

```
#include <synapseGroupInternal.h>

class SynapseGroupInternal: public SynapseGroup

public:
    // methods

    SynapseGroupInternal(
        const std::string name,
        SynapseMatrixType matrixType,
        unsigned int delaySteps,
        const WeightUpdateModels::Base* wu,
        const std::vector<double>& wuParams,
        const std::vector<Models::VarInit>& wuVarInitialisers,
        const std::vector<Models::VarInit>& wuPreVarInitialisers,
        const std::vector<Models::VarInit>& wuPostVarInitialisers,
        const PostsynapticModels::Base* ps,
        const std::vector<double>& psParams,
        const std::vector<Models::VarInit>& psVarInitialisers,
        NeuronGroupInternal* srcNeuronGroup,
        NeuronGroupInternal* trgNeuronGroup,
        const InitSparseConnectivitySnippet::Init& connectivityInitialiser,
        VarLocation defaultVarLocation,
        VarLocation defaultExtraGlobalParamLocation,
        VarLocation defaultSparseConnectivityLocation
        );
;
```

28.29.1 Inherited Members

```
public:
    // enums

    enum SpanType;

    // methods

    SynapseGroup(const SynapseGroup&);
    SynapseGroup();
    void setWUVarLocation(const std::string& varName, VarLocation loc);
    void setWUPreVarLocation(const std::string& varName, VarLocation loc);
    void setWUPostVarLocation(const std::string& varName, VarLocation loc);
    void setWUExtraGlobalParamLocation(const std::string& paramName,␣
↪VarLocation loc);
    void setPSVarLocation(const std::string& varName, VarLocation loc);
    void setPSExtraGlobalParamLocation(const std::string& paramName,␣
↪VarLocation loc);
    void setSparseConnectivityExtraGlobalParamLocation(const std::string&␣
↪paramName, VarLocation loc);
    void setInSynVarLocation(VarLocation loc);
    void setSparseConnectivityLocation(VarLocation loc);
```

```
    void setDendriticDelayLocation(VarLocation loc);
    void setMaxConnections(unsigned int maxConnections);
    void setMaxSourceConnections(unsigned int maxPostConnections);
    void setMaxDendriticDelayTimesteps(unsigned int maxDendriticDelay);
    void setSpanType(SpanType spanType);
    void setNumThreadsPerSpike(unsigned int numThreadsPerSpike);
    void setBackPropDelaySteps(unsigned int timesteps);
    const std::string& getName() const;
    SpanType getSpanType() const;
    unsigned int getNumThreadsPerSpike() const;
    unsigned int getDelaySteps() const;
    unsigned int getBackPropDelaySteps() const;
    unsigned int getMaxConnections() const;
    unsigned int getMaxSourceConnections() const;
    unsigned int getMaxDendriticDelayTimesteps() const;
    SynapseMatrixType getMatrixType() const;
    VarLocation getInSynLocation() const;
    VarLocation getSparseConnectivityLocation() const;
    VarLocation getDendriticDelayLocation() const;
    int getClusterHostID() const;
    bool isTrueSpikeRequired() const;
    bool isSpikeEventRequired() const;
    const WeightUpdateModels::Base* getWUModel() const;
    const std::vector<double>& getWUParams() const;
    const std::vector<Models::VarInit>& getWUVarInitialisers() const;
    const std::vector<Models::VarInit>& getWUPreVarInitialisers() const;
    const std::vector<Models::VarInit>& getWUPostVarInitialisers() const;
    const std::vector<double> getWUConstInitVals() const;
    const PostsynapticModels::Base* getPSModel() const;
    const std::vector<double>& getPSParams() const;
    const std::vector<Models::VarInit>& getPSVarInitialisers() const;
    const std::vector<double> getPSConstInitVals() const;
    const InitSparseConnectivitySnippet::Init& getConnectivityInitialiser()␣
↪const;
    bool isZeroCopyEnabled() const;
    VarLocation getWUVarLocation(const std::string& var) const;
    VarLocation getWUVarLocation(size_t index) const;
    VarLocation getWUPreVarLocation(const std::string& var) const;
    VarLocation getWUPreVarLocation(size_t index) const;
    VarLocation getWUPostVarLocation(const std::string& var) const;
    VarLocation getWUPostVarLocation(size_t index) const;
    VarLocation getWUExtraGlobalParamLocation(const std::string& paramName)␣
↪const;
    VarLocation getWUExtraGlobalParamLocation(size_t index) const;
    VarLocation getPSVarLocation(const std::string& var) const;
    VarLocation getPSVarLocation(size_t index) const;
    VarLocation getPSExtraGlobalParamLocation(const std::string& paramName)␣
↪const;
    VarLocation getPSExtraGlobalParamLocation(size_t index) const;
    VarLocation getSparseConnectivityExtraGlobalParamLocation(const␣
↪std::string& paramName) const;
    VarLocation getSparseConnectivityExtraGlobalParamLocation(size_t index)␣
↪const;
    bool isDendriticDelayRequired() const;
```

```
    bool isPSInitRNGRequired() const;
    bool isWUInitRNGRequired() const;
    bool isWUVarInitRequired() const;
    bool isSparseConnectivityInitRequired() const;
```

28.30 Overview

```
// namespaces

namespace CodeGenerator;
    namespace CodeGenerator::CUDA;
        namespace CodeGenerator::CUDA::Optimiser;
        namespace CodeGenerator::CUDA::PresynapticUpdateStrategy;
        namespace CodeGenerator::CUDA::Utils;
    namespace CodeGenerator::SingleThreadedCPU;
        namespace CodeGenerator::SingleThreadedCPU::Optimiser;
namespace CurrentSourceModels;
namespace InitSparseConnectivitySnippet;
namespace InitVarSnippet;
namespace Models;
namespace NeuronModels;
namespace PostsynapticModels;
namespace Snippet;
namespace Utils;
namespace WeightUpdateModels;
namespace filesystem;
namespace pygenn;
    namespace pygenn::genn_groups;
    namespace pygenn::genn_model;
    namespace pygenn::model_preprocessor;
namespace std;

// typedefs

typedef ModelSpec NNmodel;

// enums

enum FloatType;
enum MathsFunc;
enum SynapseMatrixConnectivity;
enum SynapseMatrixType;
enum SynapseMatrixWeight;
enum TimePrecision;
enum VarAccess;
enum VarLocation;

// classes

class CurrentSource;
class CurrentSourceInternal;
class ModelSpec;
```

```
class ModelSpecInternal;
class NeuronGroup;
class NeuronGroupInternal;
class SynapseGroup;
class SynapseGroupInternal;

// global functions

unsigned int binomialInverseCDF(
    double cdf,
    unsigned int n,
    double p
    );

GENN_EXPORT unsigned int binomialInverseCDF(
    double cdf,
    unsigned int n,
    double p
    );

IMPLEMENT_MODEL(CurrentSourceModels::DC);
IMPLEMENT_MODEL(CurrentSourceModels::GaussianNoise);
IMPLEMENT_SNIPPET(InitSparseConnectivitySnippet::Uninitialised);
IMPLEMENT_SNIPPET(InitSparseConnectivitySnippet::OneToOne);
IMPLEMENT_SNIPPET(InitSparseConnectivitySnippet::FixedProbability);
IMPLEMENT_SNIPPET(InitSparseConnectivitySnippet::FixedProbabilityNoAutapse);
IMPLEMENT_SNIPPET(InitVarSnippet::Uninitialised);
IMPLEMENT_SNIPPET(InitVarSnippet::Constant);
IMPLEMENT_SNIPPET(InitVarSnippet::Uniform);
IMPLEMENT_SNIPPET(InitVarSnippet::Normal);
IMPLEMENT_SNIPPET(InitVarSnippet::Exponential);
IMPLEMENT_SNIPPET(InitVarSnippet::Gamma);

template <typename S>
Models::VarInit initVar(const typename S::ParamValues& params);

template <typename S>
std::enable_if<std::is_same<typename S::ParamValues, Snippet::ValueBase<0>>::value,
 ↪ Models::VarInit>::type initVar();

Models::VarInit uninitialisedVar();

template <typename S>
InitSparseConnectivitySnippet::Init initConnectivity(const typename␣
 ↪S::ParamValues& params);

template <typename S>
std::enable_if<std::is_same<typename S::ParamValues, Snippet::ValueBase<0>>::value,
 ↪ InitSparseConnectivitySnippet::Init>::type initConnectivity();

InitSparseConnectivitySnippet::Init uninitialisedConnectivity();
IMPLEMENT_MODEL(NeuronModels::RulkovMap);
IMPLEMENT_MODEL(NeuronModels::Izhikevich);
IMPLEMENT_MODEL(NeuronModels::IzhikevichVariable);
```

```
IMPLEMENT_MODEL(NeuronModels::LIF);
IMPLEMENT_MODEL(NeuronModels::SpikeSource);
IMPLEMENT_MODEL(NeuronModels::SpikeSourceArray);
IMPLEMENT_MODEL(NeuronModels::Poisson);
IMPLEMENT_MODEL(NeuronModels::PoissonNew);
IMPLEMENT_MODEL(NeuronModels::TraubMiles);
IMPLEMENT_MODEL(NeuronModels::TraubMilesFast);
IMPLEMENT_MODEL(NeuronModels::TraubMilesAlt);
IMPLEMENT_MODEL(NeuronModels::TraubMilesNStep);
IMPLEMENT_MODEL(PostsynapticModels::ExpCurr);
IMPLEMENT_MODEL(PostsynapticModels::ExpCond);
IMPLEMENT_MODEL(PostsynapticModels::DeltaCurr);

bool operator & (
    SynapseMatrixType type,
    SynapseMatrixConnectivity connType
    );

bool operator & (
    SynapseMatrixType type,
    SynapseMatrixWeight weightType
    );

bool operator & (
    VarLocation locA,
    VarLocation locB
    );

IMPLEMENT_MODEL(WeightUpdateModels::StaticPulse);
IMPLEMENT_MODEL(WeightUpdateModels::StaticPulseDendriticDelay);
IMPLEMENT_MODEL(WeightUpdateModels::StaticGraded);
IMPLEMENT_MODEL(WeightUpdateModels::PiecewiseSTDP);

// macros

#define CHECK_CUDA_ERRORS(call)
#define CHECK_CU_ERRORS(call)

#define DECLARE_MODEL(      TYPE,      NUM_PARAMS,      NUM_VARS      )

#define DECLARE_SNIPPET(      TYPE,      NUM_PARAMS      )

#define DECLARE_WEIGHT_UPDATE_MODEL(      TYPE,      NUM_PARAMS,
↪NUM_VARS,      NUM_PRE_VARS,      NUM_POST_VARS      )

#define IMPLEMENT_MODEL(TYPE)
#define IMPLEMENT_SNIPPET(TYPE)
#define NO_DELAY
#define SET_ADDITIONAL_INPUT_VARS(...)
#define SET_APPLY_INPUT_CODE(APPLY_INPUT_CODE)
#define SET_CALC_MAX_COL_LENGTH_FUNC(FUNC)
#define SET_CALC_MAX_ROW_LENGTH_FUNC(FUNC)
#define SET_CODE(CODE)
#define SET_CURRENT_CONVERTER_CODE(CURRENT_CONVERTER_CODE)
```

```
#define SET_DECAY_CODE(DECAY_CODE)
#define SET_DERIVED_PARAMS(...)
#define SET_EVENT_CODE(EVENT_CODE)
#define SET_EVENT_THRESHOLD_CONDITION_CODE(EVENT_THRESHOLD_CONDITION_CODE)
#define SET_EXTRA_GLOBAL_PARAMS(...)
#define SET_EXTRA_GLOBAL_PARAMS(...)
#define SET_INJECTION_CODE(INJECTION_CODE)
#define SET_LEARN_POST_CODE(LEARN_POST_CODE)
#define SET_LEARN_POST_SUPPORT_CODE(LEARN_POST_SUPPORT_CODE)
#define SET_MAX_COL_LENGTH(MAX_COL_LENGTH)
#define SET_MAX_ROW_LENGTH(MAX_ROW_LENGTH)
#define SET_NEEDS_AUTO_REFRACTORY(AUTO_REFRACTORY_REQUIRED)
#define SET_NEEDS_POST_SPIKE_TIME(POST_SPIKE_TIME_REQUIRED)
#define SET_NEEDS_PRE_SPIKE_TIME(PRE_SPIKE_TIME_REQUIRED)
#define SET_PARAM_NAMES(...)
#define SET_POST_SPIKE_CODE(POST_SPIKE_CODE)
#define SET_POST_VARS(...)
#define SET_PRE_SPIKE_CODE(PRE_SPIKE_CODE)
#define SET_PRE_VARS(...)
#define SET_RESET_CODE(RESET_CODE)
#define SET_ROW_BUILD_CODE(CODE)
#define SET_ROW_BUILD_STATE_VARS(...)
#define SET_SIM_CODE(SIM_CODE)
#define SET_SIM_CODE(SIM_CODE)
#define SET_SIM_SUPPORT_CODE(SIM_SUPPORT_CODE)
#define SET_SUPPORT_CODE(SUPPORT_CODE)
#define SET_SUPPORT_CODE(SUPPORT_CODE)
#define SET_SYNAPSE_DYNAMICS_CODE(SYNAPSE_DYNAMICS_CODE)
#define SET_SYNAPSE_DYNAMICS_SUPPORT_CODE(SYNAPSE_DYNAMICS_SUPPORT_CODE)
#define SET_THRESHOLD_CONDITION_CODE(THRESHOLD_CONDITION_CODE)
#define SET_VARS(...)
#define TYPE(T)
```

28.31 Detailed Documentation

28.31.1 Global Functions

```
template <typename S>
Models::VarInit initVar(const typename S::ParamValues& params)
```

Initialise a variable using an initialisation snippet.

Parameters:

S	type of variable initialisation snippet (derived from *InitVarSnippet::Base*).
params	parameters for snippet wrapped in S::ParamValues object.

Returns:

Models::VarInit object for use within model's VarValues

```
template <typename S>
std::enable_if<std::is_same<typename S::ParamValues, Snippet::ValueBase<0>>::value,
↪ Models::VarInit>::type initVar()
```

Initialise a variable using an initialisation snippet with no parameters.

Parameters:

S	type of variable initialisation snippet (derived from *InitVarSnippet::Base*).

Returns:

Models::VarInit object for use within model's VarValues

```
Models::VarInit uninitialisedVar()
```

Mark a variable as uninitialised.

This means that the backend will not generate any automatic initialization code, but will instead copy the variable from host to device during `initializeSparse` function

```
template <typename S>
InitSparseConnectivitySnippet::Init initConnectivity(const typename␣
↪S::ParamValues& params)
```

Initialise connectivity using a sparse connectivity snippet.

Parameters:

S	type of sparse connectivitiy initialisation snippet (derived from *InitSparseConnectivitySnippet::Base*).
params	parameters for snippet wrapped in S::ParamValues object.

Returns:

InitSparseConnectivitySnippet::Init object for passing to `ModelSpec::addSynapsePopulation`

```
template <typename S>
std::enable_if<std::is_same<typename S::ParamValues, Snippet::ValueBase<0>>::value,
↪ InitSparseConnectivitySnippet::Init>::type initConnectivity()
```

Initialise connectivity using a sparse connectivity snippet with no parameters.

Parameters:

S	type of sparse connectivitiy initialisation snippet (derived from *InitSparseConnectivitySnippet::Base*).

Returns:

InitSparseConnectivitySnippet::Init object for passing to `ModelSpec::addSynapsePopulation`

`InitSparseConnectivitySnippet::Init uninitialisedConnectivity()`

Mark a synapse group's sparse connectivity as uninitialised.

This means that the backend will not generate any automatic initialization code, but will instead copy the connectivity from host to device during `initializeSparse` function (and, if necessary generate any additional data structures it requires)

28.31.2 Macros

`#define NO_DELAY`

Macro used to indicate no synapse delay for the group (only one queue slot will be generated)

Global Namespace
genindex

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X