1 GeNN Documentation

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)
Dr. Esin Yavuz (contact Esin)
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 number `EP/J019690/1` - `Green Brain Project`).

Note

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

Next

2 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.

2.1 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.

2.2 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 latest release is based. If you want the most recent (but unstable) development version (which may or may not be fully functional at any given time), checkout the development branch

```
git checkout development
```

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).

2.3 Installing GeNN

Installing GeNN comprises a few simple steps to create the GeNN development environment.

(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) Define the environment variable "GENN_PATH" to point to the main GeNN directory, e.g. if you extracted/downloaded GeNN to /usr/local/GeNN, then you can add "export GENN_PATH=/usr/local/GeNN" 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 GENN_PATH "C:\Users\me\GeNN"
```

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

(iii) Add $GENN_PATH/lib/bin to your PATH variable, e.g.

```
export PATH=$PATH:$GENN_PATH/lib/bin
```

in your login script, or in windows,

```
setx PATH=%GENN_PATH%\lib\bin;%PATH%
```

(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 "custom install", and ensure that all C++ optional extras are also installed. 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 you haven't installed CUDA on your machine, 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:\path\to\cuda
```

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

Depending on the needs of your own projects, e.g., depencies on other libraries or non-standard installation paths of libraries used by GeNN, you may want to modify Makefile examples under `$GENN_PATH/userproject/xxx_project` and `$GENN_PATH/userproject/xxx_project/model` to add extra linker-, include- and compiler-flags on a per-project basis, or modify global default flags in $GENN_PATH/userproject/include/makefile_common_[win|gnu].mk.

For all makefiles there are separate makefiles for Unix-style operating systems (GNUmakefile) such as Linux or MacOS and for Windows (WINmakefile).

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 - visual studio - tools - visual studio native command prompt. You may wish to create a shortcut for this tool on the desktop, for convenience. Note that all C++ tools should have been installed during the Visual Studio install process for this to work. Alternatively one can use the make.bat scripts to build the example projects, which will attempt to setup your development environment by executing `vcvarsall.bat` which is part of every Visual Studio distribution, inside the visual studio/VC directory. For this to work properly, GeNN must be able to locate the Visual Studio install directory, which should be contained in the `VS_PATH` environment variable. You can set this variable by hand if it is not already set by the Visual C++ installer by typing:

```
setx VS_PATH "C:\Program Files (x86)\Microsoft Visual Studio 10.0"
```

Note

- The exact path and name of Visual C++ installations will vary between systems.
- Double quotation marks like in the above example are necessary whenever a path contains spaces.

GeNN also has experimental CYGWIN support. However, with the introduction of native Windows support in GeNN 1.1.3, this is not being developed further and should be considered as deprecated.

2.4 Testing Your Installation

To test your installation, follow the example in the Quickstart section. Linux and Mac users can perform a more comprehensive test by running:

```
cd $GENN_PATH/userproject && ./testprojects.sh
```

This test script may take a long while to complete, and will terminate if any errors are detected.

Top | Next

3 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. Running these complete examples should be achievable in a few minutes. The necessary steps are described below.

3.1 Running an Example Model in Unix

In order to get a quick start and run a provided model, open a shell, navigate to `GeNN/tools` and type

```
make
```

This will compile additional tools for creating and running example projects. For a first complete test, the system is best used with a full driver program such as in the Insect olfaction model example:

```
./generate_run <0 (CPU) / 1 (GPU) / n (GPU n+2)> <nAL> <nMB> <nLHI> <nLb> <gscale> <outdir> <model
       name> <OPTIONS>
```

Possible options:
DEBUG=0 or DEBUG=1 (default 0): Whether to run in a debugger,
FTYPE=DOUBLE of FTYPE=FLOAT (default FLOAT): What floating point type to use,
REUSE=0 or REUSE=1 (default 0): Whether to reuse generated connectivity from an earlier run,
CPU_ONLY=0 or CPU_ONLY=1 (default 0): Whether to compile in (CUDA independent) "CPU only" mode.

To compile `generate_run.cc`, navigate to the `userproject/MBody1_project` directory and type

```
make
```

This will generate an executable that you can invoke with, e.g.,

```
./generate_run 1 100 1000 20 100 0.0025 test1 MBody1
```

which would generate and simulate a model of the locust olfactory system with 100 projection neurons, 1000 Kenyon cells, 20 lateral horn interneurons and 100 output neurons in the mushroom body lobes.

The tool generate_run will generate connectivity matrices for the model `MBody1` and store them into files, compile and run the model on an automatically chosen GPU, using these files as inputs and output the resulting spiking activity. To fix the GPU used, replace the first argument `1` with the device number of the desired GPU plus 2, e.g., `2` for GPU 0. All input and output files will be prefixed with `test1` and will be created in a sub-directory with the name `test1_output`. More about the DEBUG flag in the debugging section . The parameter `FLOAT` will run the model in float (single precision floating point), using `DOUBLE` would use double precision. The REU↵SE parameter regulates whether previously generated files for connectivity and input should be reused (1) or files should be generated anew (0).

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 .

3.2 Running an Example Model in Windows

All interaction with GeNN programs are command-line based and hence are executed within a `cmd` window. Open a Visual Studio `cmd` window via Start: All Programs: Visual Studio: Tools: Native Tools Command Prompt, and navigate to the `userprojects\tools` directory.

```
cd %GENN_PATH%\userprojects\tools
```

Then type

```
nmake /f WINmakefile
```

to compile a number of tools that are used by the example projects to generate connectivity and inputs to model networks. Then navigate to the `userproject/MBody1_project` directory.

```
cd ..\MBody1_project
```

By typing

```
nmake /f WINmakefile
```

you can compile the `generate_run` engine that allows to run a Insect olfaction model of the insect mushroom body:

```
generate_run <0 (CPU) / 1 (GPU) / n (GPU n+2)> <nAL> <nMB> <nLHI> <nLb> <gscale> <outdir> <model
      name> <OPTIONS>
```

To invoke `generate_run.exe` type, e.g.,

```
generate_run 1 100 1000 20 100 0.0025 test1 MBody1
```

which would generate and simulate a model of the locust olfactory system with 100 projection neurons, 1000 Kenyon cells, 20 lateral horn interneurons and 100 output neurons in the mushroom body lobes.

The tool `generate_run.exe` will generate connectivity matrices for the model `MBody1` and store them into files, compile and run the model on an automatically chosen GPU, using these files as inputs and output the resulting spiking activity. To fix the GPU used, replace the first argument `1` with the device number of the desired GPU plus 2, e.g., `2` for GPU 0. All input and output files will be prefixed with `test1` and will be created in a sub-directory with the name `test1_output`. More about the DEBUG flag in the debugging section . The parameter `FLOAT` will run the model in float (single precision floating point), using `DOUBLE` would use double precision. The REUSE parameter regulates whether previously generated files for connectivity and input should be reused (1) or files should be generated anew (0).

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 .

3.3 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. Building the source code of a model simulation 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
   ```

   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 of two files `classol_sim.cc` and `map_classol.cc`). On Linux or Mac this typically consists of a GNU makefile:

   ```
   EXECUTABLE      := classol_sim
   SOURCES         := classol_sim.cc map_classol.cc
   include $(GENN_PATH)/userproject/include/makefile_common_gnu.mk
   ```

   And on Windows an MSBuild script:

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <Project DefaultTargets="Build"  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <Import Project="MBody1_CODE\generated_code.props" />
   <ItemGroup>
       <ClCompile Include="classol_sim.cc" />
       <ClCompile Include="map_classol.cc" />
   </ItemGroup>
   </Project>
   ```

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

   ```
   make clean all
   ```

   or by invoking MSbuild on Windows:

   ```
   msbuild MBody1.vcxproj /p:Configuration=Release
   ```

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

3.4 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.

   Note

   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 heterogenous values.

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

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

   Note

   The name `modelDefinition` and its parameter of type `NNmodel&` 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);
   ```

Note

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 NNmodel↩ ::addNeuronPopulation and NNmodel::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/M↩ Body1_project/model/map_classol.*` and `userproject/MBody1_project/model/classol↩ _sim.*`. In this code,

a) They define the connectivity matrices between neuron groups. (In the MBody1 example those are read from files). 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.

Note

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 `stepTimeGPU(...);` to run one time step on the GPU or `stepTimeCPU(...);` to run one on the CPU.

Note

Both GPU and CPU versions are always compiled, unless `-c` is used with genn-buildmodel to build a CPU-only model or the model uses features not supported by the CPU simulator (see **TODO**). However, mixing CPU and GPU execution does not make too much sense. Among other things, The CPU version uses the same host side memory where to results from the GPU version are copied, which would lead to collisions between what is calculated on the CPU and on the GPU (see next point). However, in certain circumstances, expert users may want to split the calculation and calculate parts (e.g. neurons) on the GPU and parts (e.g. synapses) on the CPU. In such cases the fundamental kernel and function calls contained in `stepTimeXXX` need to be used and appropriate copies of the data from the CPU to the GPU and vice versa need to be performed.

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

4 Examples

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

4.1 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 also
prepares additional synapse connectivity and input pattern data, before compiling and
executing the model.

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

nmake /f WINmakefile

for Windows users, or:

make

for Linux, Mac and other UNIX users.


USAGE
-----

generate_run <0(CPU)/1(GPU)> <n> <DIR> <MODEL>

Optional arguments:
DEBUG=0 or DEBUG=1 (default 0): Whether to run in a debugger
FTYPE=DOUBLE of FTYPE=FLOAT (default FLOAT): What floating point type to use
REUSE=0 or REUSE=1 (default 0): Whether to reuse generated connectivity from an earlier run
CPU_ONLY=0 or CPU_ONLY=1 (default 0): Whether to compile in (CUDA independent) "CPU only" mode.

For a first minimal test, the system may be used with:

generate_run.exe 1 1 outdir OneComp

for Windows users, or:

./generate_run 1 1 outdir OneComp

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. It is lso possible to use the model
with a sinusoidal input instead, by setting the input to INPRULE.

Another example of an invocation would be:

generate_run.exe 0 1 outdir OneComp FTYPE=DOUBLE CPU_ONLY=1

for Windows users, or:

./generate_run 0 1 outdir OneComp FTYPE=DOUBLE CPU_ONLY=1

for Linux, Mac and other UNIX users.
```

Izhikevich neuron model: [1]

4.2 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 also
prepares additional synapse connectivity and input pattern data, before compiling and
executing the model.

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

nmake /f WINmakefile

for Windows users, or:
```

```
make

for Linux, Mac and other UNIX users.


USAGE
-----

generate_run <0(CPU)/1(GPU)> <nPoisson> <nIzhikevich> <pConn> <gscale> <DIR> <MODEL>

Optional arguments:
DEBUG=0 or DEBUG=1 (default 0): Whether to run in a debugger
FTYPE=DOUBLE or FTYPE=FLOAT (default FLOAT): What floating point type to use
REUSE=0 or REUSE=1 (default 0): Whether to reuse generated connectivity from an earlier run
CPU_ONLY=0 or CPU_ONLY=1 (default 0): Whether to compile in (CUDA independent) "CPU only" mode.

An example invocation of generate_run is:

generate_run.exe 1 100 10 0.5 2 outdir PoissonIzh

for Windows users, or:

./generate_run 1 100 10 0.5 2 outdir PoissonIzh

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 synapse population with sparse connectivity in PoissonIzh.cc and by uncommenting
the lines following the "//SPARSE CONNECTIVITY" tag in PoissonIzh.cu and commenting the
lines following `//DENSE CONNECTIVITY`.

Another example of an invocation would be:

generate_run.exe 0 100 10 0.5 2 outdir PoissonIzh FTYPE=DOUBLE CPU_ONLY=1

for Windows users, or:

./generate_run 0 100 10 0.5 2 outdir PoissonIzh FTYPE=DOUBLE CPU_ONLY=1

for Linux, Mac and other UNIX users.
```

Izhikevich neuron model: [1]

4.3 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 nConn neurons with sparse connectivity.

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

nmake /f WINmakefile

for Windows users, or:

make

for Linux, Mac and other UNIX users.


USAGE
-----

generate_run <0(CPU)/1(GPU)/n(GPU n-2)> <nNeurons> <nConn> <gScale> <outdir> <model name> <input factor>
```

```
Mandatory arguments:
CPU/GPU: Choose whether to run the simulation on CPU (`0`), auto GPU (`1`), or GPU (n-2) (`n`).
nNeurons: Number of neurons
nConn: Number of connections per neuron
gScale: General scaling of synaptic conductances
outname: The base name of the output location and output files
model name: The name of the model to execute, as provided this would be `Izh_sparse`

Optional arguments:
DEBUG=0 or DEBUG=1 (default 0): Whether to run in a debugger
FTYPE=DOUBLE of FTYPE=FLOAT (default FLOAT): What floating point type to use
REUSE=0 or REUSE=1 (default 0): Whether to reuse generated connectivity from an earlier run
CPU_ONLY=0 or CPU_ONLY=1 (default 0): Whether to compile in (CUDA independent) "CPU only" mode.

An example invocation of generate_run is:

generate_run.exe 1 10000 1000 1 outdir Izh_sparse 1.0

for Windows users, or:

./generate_run 1 10000 1000 1 outdir Izh_sparse 1.0

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`,
debugging is switched off, and the connectivity is generated afresh (rather than
being read from existing files).

If connectivity were to be read from files, the connectivity files would have to
be in the `inputfiles` sub-directory and be named according to the names of the
synapse populations involved, e.g., `gIzh_sparse_ee` (\<variable name>=`g`
\<model name>=`Izh_sparse` \<synapse population>=`_ee`). These name conventions
are not part of the core GeNN definitions and it is the privilege (or burden)
of the user to find their own in their own versions of `generate_run`.

Another example of an invocation would be:

generate_run.exe 0 10000 1000 1 outdir Izh_sparse 1.0 FTYPE=DOUBLE DEBUG=0 CPU_ONLY=1

for Windows users, or:

./generate_run 0 10000 1000 1 outdir Izh_sparse 1.0 FTYPE=DOUBLE DEBUG=0 CPU_ONLY=1

for Linux, Mac and other UNIX users.
```

Izhikevich neuron model: [1]

4.4 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.vcxproj /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

then compile the project by typing:

make


USAGE
-----

syn_delay [CPU = 0 / GPU = 1] [directory to save output]
```

Izhikevich neuron model: [1]

4.5 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 also
prepares additional synapse connectivity and input pattern data, before compiling and
executing the model.

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

nmake /f WINmakefile

for Windows users, or:

make

for Linux, Mac and other UNIX users.


USAGE
-----

generate_run <0(CPU)/1(GPU)/n(GPU n-2)> <nAL> <nKC> <nLH> <nDN> <gScale> <DIR> <MODEL>

Mandatory parameters:
CPU/GPU: Choose whether to run the simulation on CPU (`0`), auto GPU (`1`), or GPU (n-2) (`n`).
nAL: Number of neurons in the antennal lobe (AL), the input neurons to this model
nKC: Number of Kenyon cells (KC) in the "hidden layer"
nLH: Number of lateral horn interneurons, implementing gain control
nDN: Number of decision neurons (DN) in the output layer
gScale: A general rescaling factor for snaptic strength
outname: The base name of the output location and output files
model: The name of the model to execute, as provided this would be `MBody1`

Optional arguments:
DEBUG=0 or DEBUG=1 (default 0): Whether to run in a debugger
FTYPE=DOUBLE of FTYPE=FLOAT (default FLOAT): What floating point type to use
```

```
REUSE=0 or REUSE=1 (default 0): Whether to reuse generated connectivity from an earlier run
BITMASK=0 or BITMASK=1 (default 0): Whether to use bitmasks to represent sparse PN->KC connectivity.
DELAYED_SYNAPSES=0 or DELAYED_SYNAPSES=1 (default 0): Whether to simulate delays of (5 * DT) ms on KC->DN and
CPU_ONLY=0 or CPU_ONLY=1 (default 0): Whether to compile in (CUDA independent) "CPU only" mode.

An example invocation of generate_run is:

generate_run.exe 1 100 1000 20 100 0.0025 outname MBody1

for Windows users, or:

./generate_run 1 100 1000 20 100 0.0025 outname MBody1

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 "outname"
and will be created under the "outname" directory. The model that is run is defined
in `model/MBody1.cc`, debugging is switched off, the model would be simulated using
float (single precision floating point) variables and parameters and the connectivity
and input would be generated afresh for this run.

In more details, what generate_run program does is:
a) use some other tools to generate the appropriate connectivity
   matrices and store them in files.

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 1".

Another example of an invocation would be:

generate_run.exe 0 100 1000 20 100 0.0025 outname MBody1 FTYPE=DOUBLE CPU_ONLY=1

for Windows users, or:

./generate_run 0 100 1000 20 100 0.0025 outname MBody1 FTYPE=DOUBLE CPU_ONLY=1

for Linux, Mac and other UNIX users, for using double precision floating point
and compiling and running the "CPU only" version.

Note: Optional arguments cannot contain spaces, i.e. "CPU_ONLY= 0"
will fail.

As provided, the model outputs a file `test1.out.st` that contains
the spiking activity observed in the simulation, There are two
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. 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),
doi:10.1007/s00422-005-0019-7
```

Nowotny insect olfaction model: [3]; Traub-Miles Hodgkin-Huxley neuron model: [5]

4.6 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 on the GPU 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:

nmake /f WINmakefile

for Windows users, or:

make

for Linux, Mac and other UNIX users.


USAGE
-----

generate_run <CPU=0, GPU=1> <protocol> <nPop> <totalT> <outdir>

Mandatory parameters:
GPU/CPU: Whether to use the GPU (1) or CPU (0) for the model neuron population
protocol: Which changes to apply during the run to the parameters of the "true cell"
nPop: Number of neurons in the tracking population
totalT: Time in ms how long to run the simulation
outdir: The directory in which to save results

Optional arguments:
DEBUG=0 or DEBUG=1 (default 0): Whether to run in a debugger
FTYPE=DOUBLE of FTYPE=FLOAT (default FLOAT): What floating point type to use
REUSE=0 or REUSE=1 (default 0): Whether to reuse generated connectivity from an earlier run
CPU_ONLY=0 or CPU_ONLY=1 (default 0): Whether to compile in (CUDA independent) "CPU only" mode.

An example invocation of generate_run is:

generate_run.exe 1 -1 12 200000 test1

for Windows users, or:

./generate_run 1 -1 12 200000 test1

for Linux, Mac and other UNIX users.

This will simulate nPop= 5000 Hodgkin-Huxley neurons on the GPU which will for 1000 ms be matched to a
Hodgkin-Huxley neuron where the parameter gKd is sinusoidally modulated. 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 would be:

generate_run.exe 0 -1 12 200000 test1 FTYPE=DOUBLE CPU_ONLY=1

for Windows users, or:

./generate_run 0 -1 12 200000 test1 FTYPE=DOUBLE CPU_ONLY=1

for Linux, Mac and other UNIX users.
```

Traub-Miles Hodgkin-Huxley neuron model: [5]

4.7 A neuromorphic network for generic multivariate data classification

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

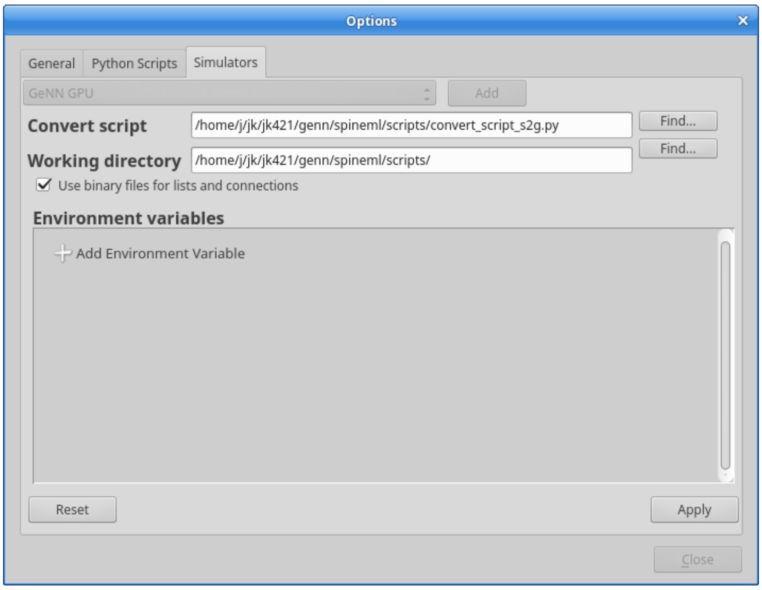
```
This project recreates using GeNN the spiking classifier design used in the paper

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

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

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

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

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 observ

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 thres

You should also note there is no option currently to run on CPU, this is not due to the demanding task, it jus
```

Previous | Top | Next

5 SpineML and SpineCreator

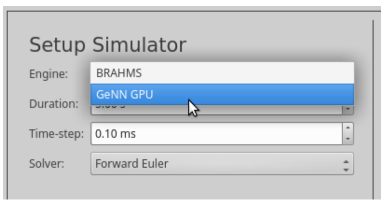
GeNN now supports simulating models built using SpineML and includes scripts to fully integrate it with the SpineCreator graphical editor on Linux, Mac and Windows. After installing GeNN using the instructions in Installation, build SpineCreator for your platform.

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". Additionally, if you are running GeNN on a 64-bit Linux system with Glibc 2.23 or 2.24 (namely Ubuntu 16.04 LTS), we recommend adding another environment variable called "LD_BIND_NOW" and setting this to "1" to work around a bug found in Glibc.

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

6 Brian interface (Brian2GeNN)

GeNN can simulate models written for the Brian simulator via the Brian2GeNN interface. 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

7 Release Notes

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.

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.

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.

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.

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.

Release Notes for GeNN v2.2.3

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

User Side Changes

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

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.

Release Notes for GeNN v2.2.2

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

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`.

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 'makefile_commin_win.mk' are now appended to, rather than being overwritten, fixing issues with custom user makefiles on Windows.

Release Notes for GeNN v2.2.1

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

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.

Release Notes for GeNN v2.2

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

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.

   Note

   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 post-synaptic 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
   ```

   Note

   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!*

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.

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.

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.

Release Notes for GeNN v2.1

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

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.

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.

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.

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.

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.

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`.

Note

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`.

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

8 User Manual

8.1 Contents

8.2 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

8.3 Defining a network model

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

```
void modelDefinition(NNmodel &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.

8.3.1 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.

8.3.2 Defining synapse populations

Synapse populations are added with the function

```
model.addSynapsePopulation<WeightUpdateModel, PostsynapticModel>(name, mType, delay,
      preName, postName, weightParamValues, weightVarValues, postsynapticParamValues, postsynapticVarValues);
```

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. the options currently are "SPARSE_GLOBA↩LG", "SPARSE_INDIVIDUALG", "DENSE_GLOBALG", "DENSE_INDIVIDUALG" or "BITMASK_GLOBALG" (see Synaptic matrix types).

- `unsigned int delay`: Synaptic delay (in multiples 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
- `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

Note

If the synapse conductance definition type is "GLOBALG" then the global value of the synapse parameters is taken from the initial value provided in `weightVarValues`.

Previous | Top | Next

8.4 Neuron models

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

- NeuronModels::RulkovMap
- NeuronModels::Izhikevich
- NeuronModels::IzhikevichVariable
- NeuronModels::SpikeSource
- NeuronModels::PoissonNew
- NeuronModels::TraubMiles
- NeuronModels::TraubMilesFast
- NeuronModels::TraubMilesAlt
- NeuronModels::TraubMilesNStep

8.4.1 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 , 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 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 NNmodel::setPrecision. The variables defined here as `NAME` can then be used in the syntax in the code string.

For example, using these macros, we can define a leaky integrator $\tau\frac{dV}{dt} = -V + I_{\rm syn}$ 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.

8.4.1.1 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); }");
```

8.4.1.2 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.

8.4.1.3 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) * $(Isyn2);\n"
             "$(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).

8.4.1.4 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.

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

8.5 Weight update models

Currently 3 predefined weight update models are available:

- WeightUpdateModels::StaticPulse
- WeightUpdateModels::StaticGraded
- WeightUpdateModels::PiecewiseSTDP

For more details about these built-in synapse models, see [2].

8.5.1 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:

- DECLARE_MODEL(TYPE, NUM_PARAMS, NUM_VARS), 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.
- 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 assigned to the $(addtoinsyn) variable. For example

```
SET_SIM_CODE(
    "\$(addtoinsyn) = $(inc);\n"
    "\$(updatelinsyn)");
```

  where "inc" is a parameter of the weight update model that defines a constant increment of the synaptic input of a post-synaptic neuron for each pre-synaptic spike. Once $(addtoinsyn) has been assigned, the $(updatelinsyn) keyword should be used to indicate that the summation of synaptic inputs can now occur. This can then be followed by updates on the internal synapse variables that may have contributed to addtoinSyn. For an example, see WeightUpdateModels::StaticPulse for a simple synapse update model and WeightUpdateModels::PiecewiseSTDP for a more complicated model that uses STDP.
- 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.

- SET_NEEDS_PRE_SPIKE_TIME(PRE_SPIKE_TIME_REQUIRED) and SET_NEEDS_POST_SPIKE_TIME(POST_SPIKE_TIME_REQUIRED) 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 can be used to manipulate any synapse variable and so implement both synaptic dynamics and learning processes.

8.6 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 Yale format. In general, this is less efficient to traverse using a GPU than the dense matrix format but does result in large memory savings for large matrices. Sparse matrices are stored in a struct named SparseProjection which contains the following members:
  1. `unsigned int connN`: number of connections in the population. This value is needed for allocation of arrays. The indices that correspond to these values are defined in a pre-to-post basis by the subsequent arrays.
  2. `unsigned int ind` (of size connN): Indices of corresponding postsynaptic neurons concatenated for each presynaptic neuron.
  3. `unsigned int *indInG` with one more entry than there are presynaptic neurons. This array defines from which index in the synapse variable array the indices in ind would correspond to the presynaptic neuron that corresponds to the index of the indInG array, with the number of connections being the size of ind. More specifically, `indIng`[i+1]-indIng[i] would give the number of postsynaptic connections for neuron i. 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 to 0th and 2nd neurons. The struct SparseProjection should have these members, with indexing from 0:

     ```
     ConnN = 4
     ind = [1 2 0 2]
     indIng = [0 2 4]
     ```

     Weight update model variables associated with the sparsely connected synaptic population will be kept in an array using this conductance for indexing. For example, a variable caled `g` will be kept in an array such as: `g`=[g_Pre0-Post1 g_pre0-post2 g_pre1-post0 g_pre1-post2] If there are no connections for a presynaptic neuron, then `g`[indIng[n]]=gp[indIng[n]+1]. See tools/gen_syns_sparse_IzhModel used in Izh_sparse project to see a working example.
- 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).

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 `pushXXXXXToDevice` 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 NNmodel::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 NNmodel::addSynapsePopulation:

- SynapseMatrixType::SPARSE_GLOBALG
- SynapseMatrixType::SPARSE_INDIVIDUALG
- SynapseMatrixType::DENSE_GLOBALG
- SynapseMatrixType::DENSE_INDIVIDUALG
- SynapseMatrixType::BITMASK_GLOBALG

8.7 Postsynaptic integration methods

There are currently 2 built-in postsynaptic integration methods:

- PostsynapticModels::ExpCond
- PostsynapticModels::DeltaCurr

8.7.1 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

8.8 Variable initialisation

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

Note

In previous versions of GeNN, weight update models state variables for synapse populations with sparse connectivity were not automatically initialised. This behaviour remains the default, but by setting

```
GENN_PREFERENCES::autoInitSparseVars=true;
```

this can be overriden.

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:

```
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

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
    ...
);
```

8.8.1 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\n"
        "{\n"
        "   normal = $(mean) + ($(gennrand_normal) * $(sd));\n"
        "} while (normal < 0.0);\n"
        "$(value) = normal;\n");

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

8.8.2 Variable initialisation modes

Once you have defined **how** your variables are going to be initialised you need to configure **where** they will be initialised and allocated. By default memory is allocated for variables on both the GPU and the host; and variables are initialised on the host as described in section Variable initialisation and then uploaded to the GPU. However, variable initialisation can also be offloaded to the GPU, potentially reducing the time spent both calculating the initial values and uploading them. To enable this functionality the following alternative modes of operation are available:

- VarMode::LOC_DEVICE_INIT_DEVICE - Variables are only allocated on the GPU (and thus initialised there), saving memory but meaning that they can't easily be copied to the host - best for internal state variables.
- VarMode::LOC_HOST_DEVICE_INIT_HOST - Variables are allocated on both the GPU and the host and are initialised on the host and automatically uploaded - the default.
- VarMode::LOC_HOST_DEVICE_INIT_DEVICE - Variables are allocated on both the GPU and the host and are initialised on the GPU - best default for new models.
- VarMode::LOC_ZERO_COPY_INIT_HOST - Variables are allocated as 'zero-copy' memory accessible to the host and GPU and initialised on the host.
- VarMode::LOC_ZERO_COPY_INIT_DEVICE - Variables are allocated as 'zero-copy' memory accessible to the host and GPU and initialised on the GPU.

Note

'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 global default using `GENN_PREFERENCES::defaultVarMode` or on a per-variable basis using one of the following functions:

- NeuronGroup::setSpikeVarMode
- NeuronGroup::setSpikeEventVarMode
- NeuronGroup::setSpikeTimeVarMode
- NeuronGroup::setVarMode
- SynapseGroup::setWUVarMode
- SynapseGroup::setPSVarMode
- SynapseGroup::setInSynVarMode

Previous | Top | Next

9 Tutorial 1

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

9.1 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 &
```

Note

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 `NNmodel`. Type in your `tenHHModel.cc` file:

```
// Model definintion file tenHHModel.cc

#include "modelSpec.h"

void modelDefinition(NNmodel &model)
{
    // definition of tenHHModel
}
```

First we should set some options:

```
GENN_PREFERENCES::defaultVarMode =
      VarMode::LOC_HOST_DEVICE_INIT_DEVICE;
```

The `defaultVarMode` option controls how model state variables will be initialised. The `VarMode::LOC_HOST_DEVICE_INIT_DEVICE` setting means that initialisation will be done on the GPU, but memory will be allocated on both the host and GPU so the values can be copied back into host memory so they can be recorded. This setting should generally be the default for new models, but section Variable initialisation modes outlines the full range of options as well as how you can control this option on a per-variable level. Now we need to fill the actual model definition. Three standard elements to the 'modelDefinition function are initialising GeNN, setting the simulation step size and setting the name of the model:

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

Note

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 μ`S`. Therefore, this defines `DT= 0.1 ms`.

Making the actual model definition makes use of the NNmodel::addNeuronPopulation and NNmodel::addSynapsePopulation member functions of the NNmodel object. The arguments to a call to NNmodel::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
);
```

Note

> 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);
```

The model definition then needs to end on calling

```
model.finalize();
```

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(NNmodel &model)
{
    // Settings
    GENN_PREFERENCES::defaultVarMode =
      VarMode::LOC_HOST_DEVICE_INIT_DEVICE;

    // definition of tenHHModel
    initGeNN();
    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);
    model.finalize();
}
```

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.

9.2 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 your environment variables `GENN_PATH` and `CUDA_PATH` are correctly configured, you should see some compile output ending in `Model build complete ...`.

9.3 User Code

GeNN will now have generated the code to simulate the model for one timestep using a function `stepTimeCPU()` (execution on CPU only) or `stepTimeGPU()` (execution on a GPU). 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 `NNmodel::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 ($\frac{1000}{DT}$ timesteps). To do so, we add after `initialize();`

```
for(int i = 0; i < (int)(1000.0 / DT); i++) {
  stepTimeGPU();
}
```

and we need to copy the result, and output it to stdout,

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

Note

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();
    for(int i = 0; i < (int)(1000.0 / DT); i++) {
        stepTimeGPU();
    }
    pullPop1StateFromDevice();
    for (int j= 0; j < 10; j++) {
        cout << VPop1[j] << " ";
        cout << mPop1[j] << " ";
        cout << hPop1[j] << " ";
        cout << nPop1[j] << endl;
    }
    return 0;
}
```

9.4 Building the simulator (Linux or Mac)

On Linux and Mac, GeNN simulations are typically built using a simple Makefile. By convention we typically call this `GNUmakefile`. Create this file and enter

```
EXECUTABLE      :=tenHHSimulation
SOURCES         :=tenHHSimulation.cc

include $(GENN_PATH)/userproject/include/makefile_common_gnu.mk
```

This defines that the final executable of this simulation is named tenHHSimulation and the simulation code is given in the file `tenHHSimulation.cc` that we completed above. Now type

```
make
```

9.5 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. Therefore create `tenHHSimulation.vcxproj` and type:

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import Project="tenHHModel_CODE\generated_code.props" />
  <ItemGroup>
    <ClCompile Include="tenHHSimulation.cc" />
  </ItemGroup>
</Project>
```

Now type

```
msbuild tenHHSimulation.vcxproj /p:Configuration=Release
```

9.6 Running the Simulation

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

```
./tenHHSimulation
```

Or on Windows with

```
tenHHSimulation
```

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
```

9.7 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 `stepTimeGPU` in `tenHHSimulation.cc` with something like this:

```
ofstream os("tenHH_output.V.dat");
for (int i= 0; i < 10000; i++) {
    stepTimeGPU();

    pullPop1StateFromDevice();
    os << t << " ";
    for (int j= 0; j < 10; j++) {
        os << VPop1[j] << " ";
    }
    os << endl;
}
os.close();
```

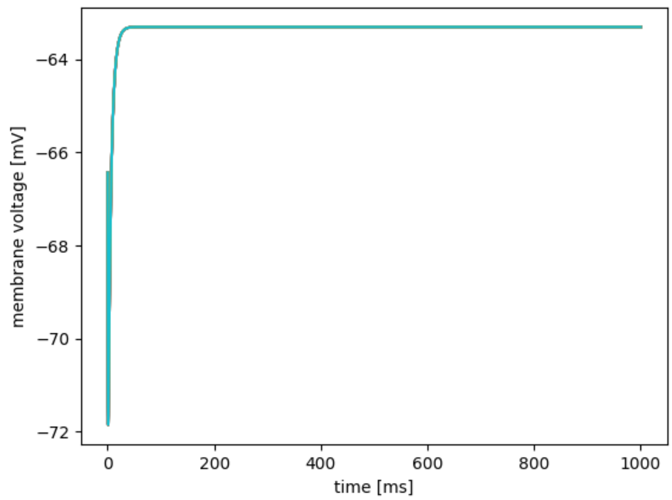
Note

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

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

10 Tutorial 2

In this tutorial we will learn to add synapsePopulations to connect neurons in neuron groups to each other with synatic 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
```

Now, we need to add a synapse group to the model that allows to connect neurons from the Pop1 group to connect to other neurons of this group. Open `tenHHRingModel.cc`, change the model name inside,

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

10.1 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(
    0.0 // 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
);
```

Note

> 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::DENSE_INDIVIDUALG, 10,
    "Pop1", "Pop1",
    {}, s_ini,
    ps_p, {});
```

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::DENSE_INDIVIDUALG means that there will be a dense connectivity matrix with seperate values for each entry.
- 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.

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. In the case of SynapseMatrixType::DENSE_INDIVIDUALG connectivity, where individual connections are present is determined by the weight update model's `g` variable which will need to be initialised in our user code. As always, the `modelDefinition` function ends on

```
model.finalize();
```

At this point our model definition file `tenHHRingModel.cc` should look like this

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

void modelDefinition(NNmodel &model)
{
    // Settings
    GENN_PREFERENCES::defaultVarMode =
      VarMode::LOC_HOST_DEVICE_INIT_DEVICE;

    // definition of tenHHRing
    initGeNN();
    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(
        0.0 // 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::DENSE_INDIVIDUALG,
      NO_DELAY,
        "Pop1", "Pop1",
        {}, s_ini,
        ps_p, {});
        model.finalize();
}
```

10.2 Defining the Detailed Synaptic Connections

Open the `tenHHRingSimulation.cc` file and update the file names of includes:

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

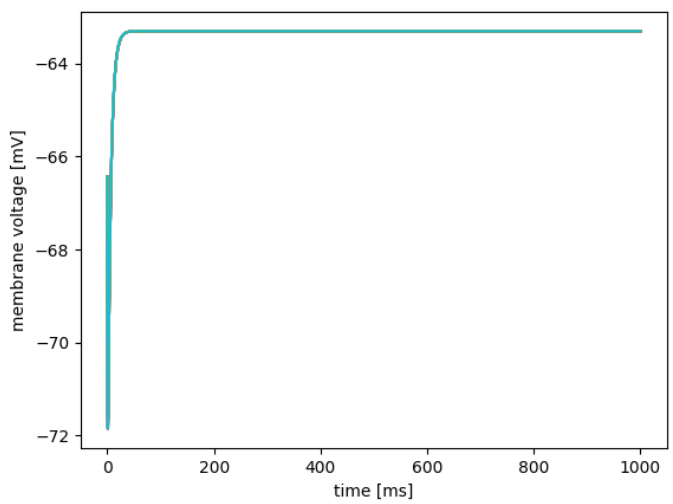
Because, after memory allocation and initialization, `gPop1self` will contain only zeros (as specified by `s_ini`), we generate the desired ring connectivity by assigning a non-zero conductivity of -0.2 μS to all synapses from neuron `i` to `i+1` (and `9` to `0` to close the ring).

```
allocateMem();
initialize();
// define the connectivity
for (int i= 0; i < 10; i++) {
    const int pre= i;
    const int post= (i+1)%10;
    gPop1self[pre*10+post]= -0.2;
}
pushPop1selfStateToDevice();
```

We can now build our new model:

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

and, after adjusting the GNUmakefile or MSBuild script to point to `tenHHRingSimulation.cc` rather than `tenHHSimulation.cc`, we can build and run our new simulator in the same way described in Tutorial 1. However if we plot the content of columns one against the subsequent 10 columns of `tenHHexample.V.dat` it looks very similar as in Tutorial 1



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

10.3 Providing initial stimuli

We can use a NeuronModels::SpikeSource to inject an initial spike during the first timestep to start spikes propagating around the ring. Firstly we 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::DENSE_INDIVIDUALG,
      NO_DELAY,
    "Stim", "Pop1",
    {}, s_ini,
    ps_p, {});
```

we can then initialise this connection's connectivity matrix in `tenHHRingSimulation.cc` file

Note

all other synapses will be initialised to zero because the synapse population used the previously-defined `s_ini` weight update initial values.

```
// define stimuli connectivity
gStimPop1[0]= -0.2;
pushStimPop1StateToDevice();
```

and finally inject a spike in the first timestep

```
if(i == 0) {
    spikeCount_Stim = 1;
    spike_Stim[0] = 0;
    pushStimSpikesToDevice();
}
```

Note

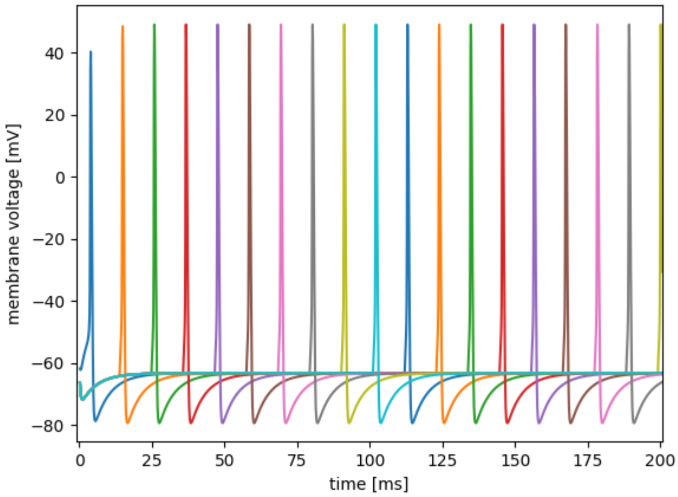
`spike_Stim[n]` is used to specify the indices of the neurons in population `Stim` spikes which should emit spikes where $n \in [0, \text{spikeCount\_Stim})$.

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

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

#include <fstream>

int main()
{
    allocateMem();
    initialize();

    // define the connectivity
    for (int i= 0; i < 10; i++) {
        const int pre= i;
        const int post= (i+1)%10;
        gPop1self[pre*10+post]= -0.2;
    }
    pushPop1selfStateToDevice();

    // define stimuli connectivity
    gStimPop1[0]= -0.2;
    pushStimPop1StateToDevice();

    ofstream os("tenHHRing_output.V.dat");
    for (int i= 0; i < 10000; i++) {
        if(i == 0) {
            spikeCount_Stim = 1;
            spike_Stim[0] = 0;
            pushStimSpikesToDevice();
        }
        stepTimeGPU();
```

```
        pullPop1StateFromDevice();
        os << t << " ";
        for (int j= 0; j < 10; j++) {
            os << VPop1[j] << " ";
        }
        s << 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

11 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.

11.1 Creating and simulating a network model

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

- initGeNN();
- NNmodel::setDT();
- NNmodel::setName();

Then add neuron populations by:

- NNmodel::addNeuronPopulation();

for each neuron population. Add synapse populations by:

- NNmodel::addSynapsePopulation();

for each synapse population.

The modelDefinition() needs to end with calling NNmodel::finalize().

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

```
void modelDefinition(NNModel &model) {
  initGeNN();
  model.setDT(0.5);
  model.setName("YourModelName");
  model.addNeuronPopulation(...);
  ...
  model.addSynapsePopulation(...);
  ...
  model.finalize();
}
```

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:

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

The following are declared by GeNN but should be initialized by the user:

- Poisson neuron offset and rates (if any)
- Connectivity matrices (if sparse)
- Neuron and synapse variables (if not initialising to the homogeneous initial value provided during `modelDefinition`)

Core functions generated by GeNN to be included in the user code include:

- `allocateMem()`
- `deviceMemAllocate()`
- `initialize()`
- `init<model name>()`
- `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>CurrentSpikesToDevice()`
- `pull<neuron name>CurrentSpikesFromDevice()`
- `push<neuron name>CurrentSpikesEventsToDevice()`

- `pull<neuron name>CurrentSpikesEventsFromDevice()`
- `copyStateToDevice()`
- `copyStateFromDevice()`
- `copySpikesToDevice()`
- `copySpikesFromDevice()`
- `copySpikesEventsToDevice()`
- `copySpikesEventsFromDevice()`
- `copyCurrentSpikesToDevice()`
- `copyCurrentSpikesFromDevice()`
- `copyCurrentSpikesEventsToDevice()`
- `copyCurrentSpikesEventsFromDevice()`
- `stepTimeCPU()`
- `stepTimeGPU()`
- `freeMem()`

Before calling the kernels, **make sure you have copied the initial values of any neuron and synapse variables initialised on the host to the GPU**. 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. Alternatively, you can directly use the CUDA memcopy functions. **Copying elements between the GPU and the host memory is very costly in terms of performance and should only be done when needed.**

11.2 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 NNmodel::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 NNmodel::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.

11.3 Working with variables in GeNN

11.3.1 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 `pullXXStatefromDevice()` and `pushXXStatetoDevice()` to copy the variables associated to a neuron population `XX` from the device into host memory and vice versa. E.g.

```
pullPopStateFromDevice();
```

would copy the C array VPop from device memory into host memory (and any other variables that the population `Pop` may have).

The user can also directly use CUDA memory copy commands independent of the provided convenience functions. The relevant device pointers for all variables that exist in host memory have the same name as the host variable but are prefixed with `d_`. For example, the copy command that would be contained in `pullPopStateFromDevice()` will look like

```
unsigned int size = sizeof(scalar) * nPop;
cudaMemcpy(VPop, d_VPop, size, cudaMemcpyDeviceToHost);
```

where `nPop` is an integer containing the population size of the `Pop` population.

Thes conventions also apply to the the variables of postsynaptic and weight update models.

Note

> 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.

11.3.2 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).
- `addtoinSyn` : This variable is used by WeightUpdateModels::Base for updating synaptic input. The way it is modified is defined using the SET_SIM_CODE or SET_EVENT_CODE macros, therefore if a user defines her own model she should update this variable to contain the input to the post-synaptic model.
- `updatelinsyn` : At the end of the synaptic update by `addtoinSyn`, final values are copied back to the d_inSyn<synapsePopulation> variables which will be used in the next step of the neuron update to provide the input to the postsynaptic neurons. This keyword designated where the changes to `addtoinSyn` have been completed and it is safe to update the summed synaptic input and write back to d_inSyn<synapsePopulation> in device memory.
- `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 $I_{\text{syn}} = g * s * (V_{\text{rev}} - V_{\text{post}})$.

Note

> The `addtoinSyn` variables from all incoming synapses are automatically summed and added to the current value of `inSyn`.

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.

11.4 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=1). 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.

Note

> 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 vcxproj 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.

Note

> 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 1' to use an output directory called outdir and to run the model on the GPU.

Previous | Top | Next

12 Credits

GeNN was created by Thomas Nowotny.

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.

Previous | Top

13 Namespace Index

13.1 Namespace List

Here is a list of all namespaces with brief descriptions:

14 Hierarchical Index

14.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

15 Class Index

15.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

16 File Index

16.1 File List

Here is a list of all files with brief descriptions:

17 Namespace Documentation

17.1 GENN_FLAGS Namespace Reference

Variables

- const unsigned int calcSynapseDynamics = 0
- const unsigned int calcSynapses = 1
- const unsigned int learnSynapsesPost = 2
- const unsigned int calcNeurons = 3

17.1.1 Variable Documentation

17.1.1.1 const unsigned int GENN_FLAGS::calcNeurons = 3

17.1.1.2 const unsigned int GENN_FLAGS::calcSynapseDynamics = 0

17.1.1.3 const unsigned int GENN_FLAGS::calcSynapses = 1

17.1.1.4 const unsigned int GENN_FLAGS::learnSynapsesPost = 2

17.2 GENN_PREFERENCES Namespace Reference

Variables

- bool optimiseBlockSize = true

  Flag for signalling whether or not block size optimisation should be performed.
- bool autoChooseDevice = true

  Flag to signal whether the GPU device should be chosen automatically.
- bool optimizeCode = false

  Request speed-optimized code, at the expense of floating-point accuracy.
- bool debugCode = false

  Request debug data to be embedded in the generated code.
- bool showPtxInfo = false

  Request that PTX assembler information be displayed for each CUDA kernel during compilation.
- bool buildSharedLibrary = false

  Should generated code and Makefile build into a shared library e.g. for use in SpineML simulator.
- bool autoInitSparseVars = false

  Previously, variables associated with sparse synapse populations were not automatically initialised. If this flag is set this now occurs in the initMODEL_NAME function and copyStateToDevice is deferred until here.
- VarMode defaultVarMode = VarMode::LOC_HOST_DEVICE_INIT_HOST

  What is the default behaviour for model state variables? Historically, everything was allocated on both host AND device and initialised on HOST.
- double asGoodAsZero = 1e-19

  Global variable that is used when detecting close to zero values, for example when setting sparse connectivity from a dense matrix.
- int defaultDevice = 0
- unsigned int neuronBlockSize = 32

  default GPU device; used to determine which GPU to use if chooseDevice is 0 (off)
- unsigned int synapseBlockSize = 32
- unsigned int learningBlockSize = 32
- unsigned int synapseDynamicsBlockSize = 32
- unsigned int initBlockSize = 32
- unsigned int initSparseBlockSize = 32
- unsigned int autoRefractory = 1

  Flag for signalling whether spikes are only reported if thresholdCondition changes from false to true (autoRefractory == 1) or spikes are emitted whenever thresholdCondition is true no matter what.%.
- std::string userCxxFlagsWIN = ""

  Allows users to set specific C++ compiler options they may want to use for all host side code (used for windows platforms)
- std::string userCxxFlagsGNU = ""

  Allows users to set specific C++ compiler options they may want to use for all host side code (used for unix based platforms)
- std::string userNvccFlags = ""

  Allows users to set specific nvcc compiler options they may want to use for all GPU code (identical for windows and unix platforms)

17.2.1 Variable Documentation

17.2.1.1 double GENN_PREFERENCES::asGoodAsZero = 1e-19

Global variable that is used when detecting close to zero values, for example when setting sparse connectivity from a dense matrix.

17.2.1.2 bool GENN_PREFERENCES::autoChooseDevice = true

Flag to signal whether the GPU device should be chosen automatically.

17.2.1.3 bool GENN_PREFERENCES::autoInitSparseVars = false

Previously, variables associated with sparse synapse populations were not automatically initialised. If this flag is set this now occurs in the initMODEL_NAME function and copyStateToDevice is deferred until here.

17.2.1.4 unsigned int GENN_PREFERENCES::autoRefractory = 1

Flag for signalling whether spikes are only reported if thresholdCondition changes from false to true (autoRefractory == 1) or spikes are emitted whenever thresholdCondition is true no matter what.%.

Flag for signalling whether spikes are only reported if thresholdCondition changes from false to true (autoRefractory == 1) or spikes are emitted whenever thresholdCondition is true no matter what.

17.2.1.5 bool GENN_PREFERENCES::buildSharedLibrary = false

Should generated code and Makefile build into a shared library e.g. for use in SpineML simulator.

17.2.1.6 bool GENN_PREFERENCES::debugCode = false

Request debug data to be embedded in the generated code.

17.2.1.7 int GENN_PREFERENCES::defaultDevice = 0

17.2.1.8 VarMode GENN_PREFERENCES::defaultVarMode = VarMode::LOC_HOST_DEVICE_INIT_HOST

What is the default behaviour for model state variables? Historically, everything was allocated on both host AND device and initialised on HOST.

17.2.1.9 unsigned int GENN_PREFERENCES::initBlockSize = 32

17.2.1.10 unsigned int GENN_PREFERENCES::initSparseBlockSize = 32

17.2.1.11 unsigned int GENN_PREFERENCES::learningBlockSize = 32

17.2.1.12 unsigned int GENN_PREFERENCES::neuronBlockSize = 32

default GPU device; used to determine which GPU to use if chooseDevice is 0 (off)

17.2.1.13 bool GENN_PREFERENCES::optimiseBlockSize = true

Flag for signalling whether or not block size optimisation should be performed.

17.2.1.14 bool GENN_PREFERENCES::optimizeCode = false

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

17.2.1.15 bool GENN_PREFERENCES::showPtxInfo = false

Request that PTX assembler information be displayed for each CUDA kernel during compilation.

17.2.1.16 unsigned int GENN_PREFERENCES::synapseBlockSize = 32

17.2.1.17 unsigned int GENN_PREFERENCES::synapseDynamicsBlockSize = 32

17.2.1.18 std::string GENN_PREFERENCES::userCxxFlagsGNU = ""

Allows users to set specific C++ compiler options they may want to use for all host side code (used for unix based platforms)

17.2.1.19 std::string GENN_PREFERENCES::userCxxFlagsWIN = ""

Allows users to set specific C++ compiler options they may want to use for all host side code (used for windows platforms)

17.2.1.20 std::string GENN_PREFERENCES::userNvccFlags = ""

Allows users to set specific nvcc compiler options they may want to use for all GPU code (identical for windows and unix platforms)

17.3 InitVarSnippet Namespace Reference

Base class for all value initialisation snippets.

Classes

- class Base
- class Constant

  Initialises variable to a constant value.
- class Exponential

  Initialises variable by sampling from the exponential distribution.
- class Normal

  Initialises variable by sampling from the normal distribution.
- class Uniform

  Initialises variable by sampling from the uniform distribution.
- class Uninitialised

  Used to mark variables as uninitialised - no initialisation code will be run.

17.3.1 Detailed Description

Base class for all value initialisation snippets.

17.4 NeuronModels Namespace Reference

Classes

- class Base

  Base class for all neuron models.
- class Izhikevich

  Izhikevich neuron with fixed parameters [1].
- class IzhikevichVariable

  Izhikevich neuron with variable parameters [1].
- class LegacyWrapper

  Wrapper around legacy weight update models stored in nModels array of neuronModel objects.
- class Poisson

  Poisson neurons.

- class PoissonNew

    Poisson neurons.
- class RulkovMap

    Rulkov Map neuron.
- class SpikeSource

    Empty neuron which allows setting spikes from external sources.
- class TraubMiles

    Hodgkin-Huxley neurons with Traub & Miles algorithm.
- class TraubMilesAlt

    Hodgkin-Huxley neurons with Traub & Miles algorithm.
- class TraubMilesFast

    Hodgkin-Huxley neurons with Traub & Miles algorithm: Original fast implementation, using 25 inner iterations.
- class TraubMilesNStep

    Hodgkin-Huxley neurons with Traub & Miles algorithm.

17.5 NewModels Namespace Reference

Classes

- class Base

    Base class for all models.
- class LegacyWrapper

    Wrapper around old-style models stored in global arrays and referenced by index.
- class VarInit
- class VarInitContainerBase
- class VarInitContainerBase< 0 >

17.6 PostsynapticModels Namespace Reference

Classes

- class Base

    Base class for all postsynaptic models.
- class DeltaCurr

    Simple delta current synapse.
- class ExpCond

    Exponential decay with synaptic input treated as a conductance value.
- class LegacyWrapper

17.7 Snippet Namespace Reference

Classes

- class Base

    Base class for all code snippets.
- class ValueBase
- class ValueBase< 0 >

17.7.1 Detailed Description

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.

17.8 StandardGeneratedSections Namespace Reference

Functions

- void neuronOutputInit (CodeStream &os, const NeuronGroup &ng, const std::string &devPrefix)
- void neuronLocalVarInit (CodeStream &os, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const std::string &devPrefix, const std::string &localID)
- void neuronLocalVarWrite (CodeStream &os, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const std::string &devPrefix, const std::string &localID)
- void neuronSpikeEventTest (CodeStream &os, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::string &localID, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

17.8.1 Function Documentation

17.8.1.1 void StandardGeneratedSections::neuronLocalVarInit (CodeStream & *os*, const NeuronGroup & *ng*, const VarNameIterCtx & *nmVars*, const std::string & *devPrefix*, const std::string & *localID*)

17.8.1.2 void StandardGeneratedSections::neuronLocalVarWrite (CodeStream & *os*, const NeuronGroup & *ng*, const VarNameIterCtx & *nmVars*, const std::string & *devPrefix*, const std::string & *localID*)

17.8.1.3 void StandardGeneratedSections::neuronOutputInit (CodeStream & *os*, const NeuronGroup & *ng*, const std::string & *devPrefix*)

17.8.1.4 void StandardGeneratedSections::neuronSpikeEventTest (CodeStream & *os*, const NeuronGroup & *ng*, const VarNameIterCtx & *nmVars*, const ExtraGlobalParamNameIterCtx & *nmExtraGlobalParams*, const std::string & *localID*, const std::vector< FunctionTemplate > *functions*, const std::string & *ftype*, const std::string & *rng*)

17.9 StandardSubstitutions Namespace Reference

Functions

- void postSynapseApplyInput (std::string &psCode, const SynapseGroup *sg, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const DerivedParamNameIterCtx &nmDerivedParams, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

  Applies standard set of variable substitutions to postsynaptic model's "apply input" code.
- void postSynapseDecay (std::string &pdCode, const SynapseGroup *sg, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const DerivedParamNameIterCtx &nmDerivedParams, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

  Name of the RNG to use for any probabilistic operations.
- void neuronThresholdCondition (std::string &thCode, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const DerivedParamNameIterCtx &nmDerivedParams, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

  Applies standard set of variable substitutions to neuron model's "threshold condition" code.
- void neuronSim (std::string &sCode, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const DerivedParamNameIterCtx &nmDerivedParams, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)
- void neuronSpikeEventCondition (std::string &eCode, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)
- void neuronReset (std::string &rCode, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const DerivedParamNameIterCtx &nmDerivedParams, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

- void weightUpdateThresholdCondition (std::string &eCode, const SynapseGroup &sg, const DerivedParamNameIterCtx &wuDerivedParams, const ExtraGlobalParamNameIterCtx &wuExtraGlobalParams, const string &preIdx, const string &postIdx, const string &devPrefix, const std::vector< FunctionTemplate > functions, const std::string &ftype)
- void weightUpdateSim (std::string &wCode, const SynapseGroup &sg, const VarNameIterCtx &wuVars, const DerivedParamNameIterCtx &wuDerivedParams, const ExtraGlobalParamNameIterCtx &wuExtraGlobalParams, const string &preIdx, const string &postIdx, const string &devPrefix, const std::vector< FunctionTemplate > functions, const std::string &ftype)
- void weightUpdateDynamics (std::string &SDcode, const SynapseGroup *sg, const VarNameIterCtx &wuVars, const DerivedParamNameIterCtx &wuDerivedParams, const ExtraGlobalParamNameIterCtx &wuExtraGlobalParams, const string &preIdx, const string &postIdx, const string &devPrefix, const std::vector< FunctionTemplate > functions, const std::string &ftype)
- void weightUpdatePostLearn (std::string &code, const SynapseGroup *sg, const DerivedParamNameIterCtx &wuDerivedParams, const ExtraGlobalParamNameIterCtx &wuExtraGlobalParams, const string &preIdx, const string &postIdx, const string &devPrefix, const std::vector< FunctionTemplate > functions, const std::string &ftype)
- std::string initVariable (const NewModels::VarInit &varInit, const std::string &varName, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

17.9.1 Function Documentation

17.9.1.1 std::string StandardSubstitutions::initVariable (const **NewModels::VarInit** & *varInit,* const std::string & *varName,* const std::vector< **FunctionTemplate** > *functions,* const std::string & *ftype,* const std::string & *rng*)

17.9.1.2 void StandardSubstitutions::neuronReset (std::string & *rCode,* const **NeuronGroup** & *ng,* const **VarNameIterCtx** & *nmVars,* const **DerivedParamNameIterCtx** & *nmDerivedParams,* const **ExtraGlobalParamNameIterCtx** & *nmExtraGlobalParams,* const std::vector< **FunctionTemplate** > *functions,* const std::string & *ftype,* const std::string & *rng*)

17.9.1.3 void StandardSubstitutions::neuronSim (std::string & *sCode,* const **NeuronGroup** & *ng,* const **VarNameIterCtx** & *nmVars,* const **DerivedParamNameIterCtx** & *nmDerivedParams,* const **ExtraGlobalParamNameIterCtx** & *nmExtraGlobalParams,* const std::vector< **FunctionTemplate** > *functions,* const std::string & *ftype,* const std::string & *rng*)

17.9.1.4 void StandardSubstitutions::neuronSpikeEventCondition (std::string & *eCode,* const **NeuronGroup** & *ng,* const **VarNameIterCtx** & *nmVars,* const **ExtraGlobalParamNameIterCtx** & *nmExtraGlobalParams,* const std::vector< **FunctionTemplate** > *functions,* const std::string & *ftype,* const std::string & *rng*)

17.9.1.5 void StandardSubstitutions::neuronThresholdCondition (std::string & *thCode,* const **NeuronGroup** & *ng,* const **VarNameIterCtx** & *nmVars,* const **DerivedParamNameIterCtx** & *nmDerivedParams,* const **ExtraGlobalParamNameIterCtx** & *nmExtraGlobalParams,* const std::vector< **FunctionTemplate** > *functions,* const std::string & *ftype,* const std::string & *rng*)

Applies standard set of variable substitutions to neuron model's "threshold condition" code.

17.9.1.6 void StandardSubstitutions::postSynapseApplyInput (std::string & *psCode,* const **SynapseGroup** * *sg,* const **NeuronGroup** & *ng,* const **VarNameIterCtx** & *nmVars,* const **DerivedParamNameIterCtx** & *nmDerivedParams,* const **ExtraGlobalParamNameIterCtx** & *nmExtraGlobalParams,* const std::vector< **FunctionTemplate** > *functions,* const std::string & *ftype,* const std::string & *rng*)

Applies standard set of variable substitutions to postsynaptic model's "apply input" code.

Parameters

psCode	the code string to work on
ng	Synapse group postsynaptic model is used in
nmVars	The postsynaptic neuron group

Parameters

ftype	Appropriate array of platform-specific function templates used to implement platform-specific functions e.g. gennrand_uniform
rng	Floating point type used by model e.g. "float"

17.9.1.7 void StandardSubstitutions::postSynapseDecay (std::string & *pdCode*, const SynapseGroup ∗ *sg*, const NeuronGroup & *ng*, const VarNameIterCtx & *nmVars*, const DerivedParamNameIterCtx & *nmDerivedParams*, const ExtraGlobalParamNameIterCtx & *nmExtraGlobalParams*, const std::vector< FunctionTemplate > *functions*, const std::string & *ftype*, const std::string & *rng*)

Name of the RNG to use for any probabilistic operations.

Applies standard set of variable substitutions to postsynaptic model's "decay" code

17.9.1.8 void StandardSubstitutions::weightUpdateDynamics (std::string & *SDcode*, const SynapseGroup ∗ *sg*, const VarNameIterCtx & *wuVars*, const DerivedParamNameIterCtx & *wuDerivedParams*, const ExtraGlobalParamNameIterCtx & *wuExtraGlobalParams*, const string & *preIdx*, const string & *postIdx*, const string & *devPrefix*, const std::vector< FunctionTemplate > *functions*, const std::string & *ftype*)

Parameters

preIdx	index of the pre-synaptic neuron to be accessed for _pre variables; differs for different Span)
postIdx	index of the post-synaptic neuron to be accessed for _post variables; differs for different Span)

17.9.1.9 void StandardSubstitutions::weightUpdatePostLearn (std::string & *code*, const SynapseGroup ∗ *sg*, const DerivedParamNameIterCtx & *wuDerivedParams*, const ExtraGlobalParamNameIterCtx & *wuExtraGlobalParams*, const string & *preIdx*, const string & *postIdx*, const string & *devPrefix*, const std::vector< FunctionTemplate > *functions*, const std::string & *ftype*)

Parameters

preIdx	index of the pre-synaptic neuron to be accessed for _pre variables; differs for different Span)
postIdx	index of the post-synaptic neuron to be accessed for _post variables; differs for different Span)

17.9.1.10 void StandardSubstitutions::weightUpdateSim (std::string & *wCode*, const SynapseGroup & *sg*, const VarNameIterCtx & *wuVars*, const DerivedParamNameIterCtx & *wuDerivedParams*, const ExtraGlobalParamNameIterCtx & *wuExtraGlobalParams*, const string & *preIdx*, const string & *postIdx*, const string & *devPrefix*, const std::vector< FunctionTemplate > *functions*, const std::string & *ftype*)

Parameters

preIdx	index of the pre-synaptic neuron to be accessed for _pre variables; differs for different Span)
postIdx	index of the post-synaptic neuron to be accessed for _post variables; differs for different Span)

17.9.1.11 void StandardSubstitutions::weightUpdateThresholdCondition (std::string & *eCode*, const SynapseGroup & *sg*, const DerivedParamNameIterCtx & *wuDerivedParams*, const ExtraGlobalParamNameIterCtx & *wuExtraGlobalParams*, const string & *preIdx*, const string & *postIdx*, const string & *devPrefix*, const std::vector< FunctionTemplate > *functions*, const std::string & *ftype*)

Parameters

preIdx	index of the pre-synaptic neuron to be accessed for _pre variables; differs for different Span)

Parameters

postIdx	index of the post-synaptic neuron to be accessed for _post variables; differs for different Span)

17.10 WeightUpdateModels Namespace Reference

Classes

- class Base

  Base class for all weight update models.
- class LegacyWrapper

  Wrapper around legacy weight update models stored in weightUpdateModels array of weightUpdateModel objects.
- class PiecewiseSTDP

  This is a simple STDP rule including a time delay for the finite transmission speed of the synapse.
- class StaticGraded

  Graded-potential, static synapse.
- class StaticPulse

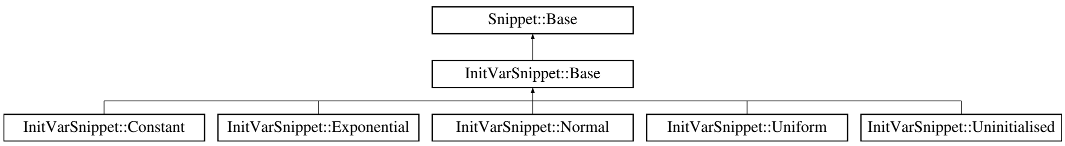
  Pulse-coupled, static synapse.

18 Class Documentation

18.1 InitVarSnippet::Base Class Reference

`#include <initVarSnippet.h>`

Inheritance diagram for InitVarSnippet::Base:



Public Member Functions

- virtual std::string getCode () const

Additional Inherited Members

18.1.1 Member Function Documentation

18.1.1.1 virtual std::string InitVarSnippet::Base::getCode ( ) const `[inline]`, `[virtual]`

The documentation for this class was generated from the following file:

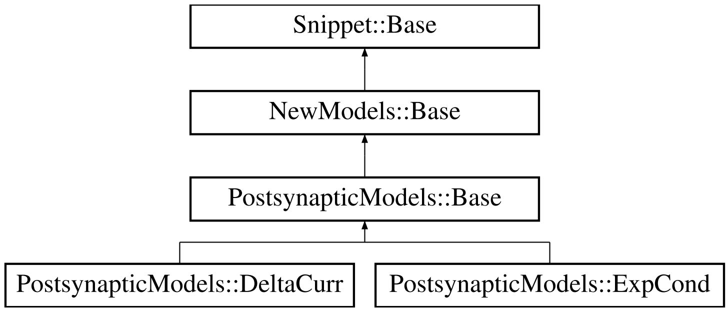
- initVarSnippet.h

18.2 PostsynapticModels::Base Class Reference

Base class for all postsynaptic models.

`#include <newPostsynapticModels.h>`

Inheritance diagram for PostsynapticModels::Base:



Public Member Functions

- virtual std::string getDecayCode () const
- virtual std::string getApplyInputCode () const
- virtual std::string getSupportCode () const

Additional Inherited Members

18.2.1 Detailed Description

Base class for all postsynaptic models.

18.2.2 Member Function Documentation

18.2.2.1 virtual std::string PostsynapticModels::Base::getApplyInputCode ( ) const [inline], [virtual]

Reimplemented in PostsynapticModels::DeltaCurr, and PostsynapticModels::ExpCond.

18.2.2.2 virtual std::string PostsynapticModels::Base::getDecayCode ( ) const [inline], [virtual]

Reimplemented in PostsynapticModels::ExpCond.

18.2.2.3 virtual std::string PostsynapticModels::Base::getSupportCode ( ) const [inline], [virtual]

The documentation for this class was generated from the following file:

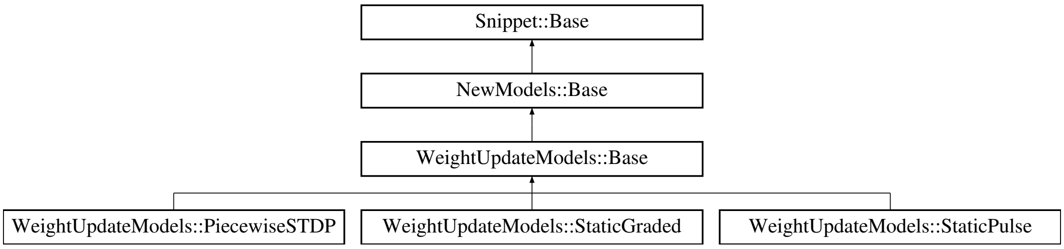
- newPostsynapticModels.h

18.3 WeightUpdateModels::Base Class Reference

Base class for all weight update models.

`#include <newWeightUpdateModels.h>`

Inheritance diagram for WeightUpdateModels::Base:



Public Member Functions

- 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.
- 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.
- virtual std::string getLearnPostSupportCode () const

    Gets support code to be made available within learnSynapsesPost kernel/function.
- virtual std::string getSynapseDynamicsSuppportCode () const

    Gets support code to be made available within the synapse dynamics kernel/function.
- virtual StringPairVec getExtraGlobalParams () const
- virtual bool isPreSpikeTimeRequired () const

    Whether presynaptic spike times are needed or not.
- virtual bool isPostSpikeTimeRequired () const

    Whether postsynaptic spike times are needed or not.

Additional Inherited Members

18.3.1 Detailed Description

Base class for all weight update models.

18.3.2 Member Function Documentation

18.3.2.1 virtual std::string WeightUpdateModels::Base::getEventCode ( ) const [inline], [virtual]

Gets code run when events (all the instances where event threshold condition is met) are received.

Reimplemented in WeightUpdateModels::StaticGraded.

18.3.2.2 virtual std::string WeightUpdateModels::Base::getEventThresholdConditionCode ( ) const [inline], [virtual]

Gets codes to test for events.

Reimplemented in WeightUpdateModels::StaticGraded.

18.3.2.3 virtual StringPairVec WeightUpdateModels::Base::getExtraGlobalParams ( ) const [inline], [virtual]

Gets names and types (as strings) of additional per-population parameters for the weight update model.

18.3.2.4 virtual std::string WeightUpdateModels::Base::getLearnPostCode ( ) const [inline], [virtual]

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.

Reimplemented in WeightUpdateModels::PiecewiseSTDP.

18.3.2.5 virtual std::string WeightUpdateModels::Base::getLearnPostSupportCode ( ) const [inline], [virtual]

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.

18.3.2.6 virtual std::string WeightUpdateModels::Base::getSimCode ( ) const [inline], [virtual]

Gets simulation code run when 'true' spikes are received.

Reimplemented in WeightUpdateModels::PiecewiseSTDP, and WeightUpdateModels::StaticPulse.

18.3.2.7 virtual std::string WeightUpdateModels::Base::getSimSupportCode ( ) const [inline], [virtual]

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

18.3.2.8 virtual std::string WeightUpdateModels::Base::getSynapseDynamicsCode ( ) const [inline], [virtual]

Gets code for synapse dynamics which are independent of spike detection.

18.3.2.9 virtual std::string WeightUpdateModels::Base::getSynapseDynamicsSuppportCode ( ) const [inline], [virtual]

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.

18.3.2.10 virtual bool WeightUpdateModels::Base::isPostSpikeTimeRequired ( ) const [inline], [virtual]

Whether postsynaptic spike times are needed or not.

Reimplemented in WeightUpdateModels::PiecewiseSTDP.

18.3.2.11 virtual bool WeightUpdateModels::Base::isPreSpikeTimeRequired ( ) const [inline], [virtual]

Whether presynaptic spike times are needed or not.

Reimplemented in WeightUpdateModels::PiecewiseSTDP.

The documentation for this class was generated from the following file:

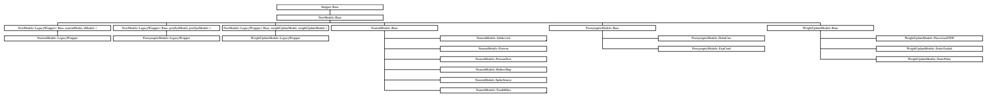
- newWeightUpdateModels.h

18.4 NewModels::Base Class Reference

Base class for all models.

```
#include <newModels.h>
```

Inheritance diagram for NewModels::Base:



Public Types

- typedef std::vector< std::pair< std::string, std::string > > StringPairVec
- typedef std::vector< std::pair< std::string, std::pair< std::string, double > > > NameTypeValVec

Public Member Functions

- virtual StringPairVec getVars () const

  Gets names and types (as strings) of model variables.
- size_t getVarIndex (const std::string &varName) const

  Find the index of a named variable.

18.4.1 Detailed Description

Base class for all models.

18.4.2 Member Typedef Documentation

18.4.2.1 typedef std::vector<std::pair<std::string, std::pair<std::string, double> > > **NewModels::Base::NameTypeValVec**

18.4.2.2 typedef std::vector<std::pair<std::string, std::string> > **NewModels::Base::StringPairVec**

18.4.3 Member Function Documentation

18.4.3.1 size_t NewModels::Base::getVarIndex (const std::string & *varName*) const [inline]

Find the index of a named variable.

18.4.3.2 virtual **StringPairVec** NewModels::Base::getVars ( ) const [inline], [virtual]

Gets names and types (as strings) of model variables.

Reimplemented in NeuronModels::TraubMiles, NeuronModels::PoissonNew, NeuronModels::Poisson, WeightUpdateModels::PiecewiseSTDP, NewModels::LegacyWrapper< Base, neuronModel, nModels >, NewModels::LegacyWrapper< Base, weightUpdateModel, weightUpdateModels >, NewModels::LegacyWrapper< Base, postSynModel, postSynModels >, NeuronModels::IzhikevichVariable, NeuronModels::Izhikevich, WeightUpdateModels::StaticGraded, NeuronModels::RulkovMap, and WeightUpdateModels::StaticPulse.

The documentation for this class was generated from the following file:

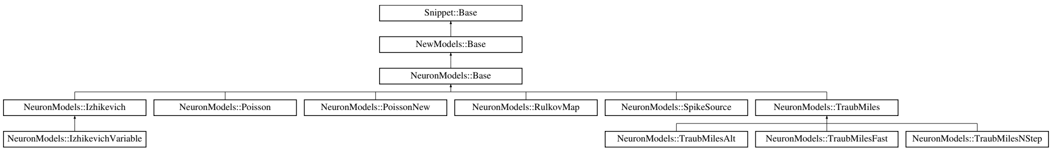
- newModels.h

18.5 NeuronModels::Base Class Reference

Base class for all neuron models.

```
#include <newNeuronModels.h>
```

Inheritance diagram for NeuronModels::Base:



Public Member Functions

- virtual std::string getSimCode () const

  Gets the code that defines the execution of one timestep of integration of the neuron model.
- virtual std::string getThresholdConditionCode () const

  Gets code which defines the condition for a true spike in the described neuron model.
- 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.
- virtual NewModels::Base::StringPairVec getExtraGlobalParams () const
- virtual NewModels::Base::NameTypeValVec getAdditionalInputVars () const

Additional Inherited Members

18.5.1 Detailed Description

Base class for all neuron models.

18.5.2 Member Function Documentation

18.5.2.1 virtual **NewModels::Base::NameTypeValVec** NeuronModels::Base::getAdditionalInputVars ( ) const [inline], [virtual]

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

18.5.2.2 virtual **NewModels::Base::StringPairVec** NeuronModels::Base::getExtraGlobalParams ( ) const [inline], [virtual]

Gets names and types (as strings) of additional per-population parameters for the weight update model.

Reimplemented in NeuronModels::Poisson.

18.5.2.3 virtual std::string NeuronModels::Base::getResetCode ( ) const [inline], [virtual]

Gets code that defines the reset action taken after a spike occurred. This can be empty.

18.5.2.4 virtual std::string NeuronModels::Base::getSimCode ( ) const [inline], [virtual]

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.

Reimplemented in NeuronModels::TraubMilesNStep, NeuronModels::TraubMilesAlt, NeuronModels::TraubMiles←Fast, NeuronModels::TraubMiles, NeuronModels::PoissonNew, NeuronModels::Poisson, NeuronModels::Izhikevich, and NeuronModels::RulkovMap.

18.5.2.5 virtual std::string NeuronModels::Base::getSupportCode ( ) const [inline], [virtual]

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.

18.5.2.6 virtual std::string NeuronModels::Base::getThresholdConditionCode ( ) const [inline], [virtual]

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

Reimplemented in NeuronModels::TraubMiles, NeuronModels::PoissonNew, NeuronModels::Poisson, Neuron←Models::SpikeSource, NeuronModels::Izhikevich, and NeuronModels::RulkovMap.

The documentation for this class was generated from the following file:

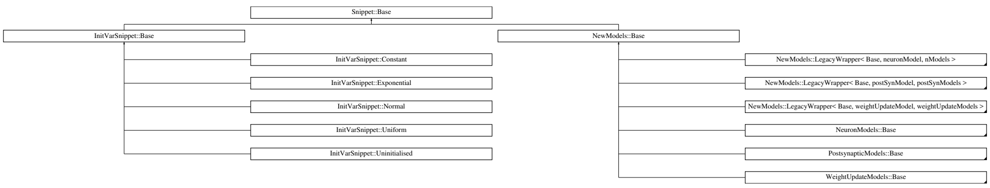
- newNeuronModels.h

18.6 Snippet::Base Class Reference

Base class for all code snippets.

`#include <snippet.h>`

Inheritance diagram for Snippet::Base:



Public Types

- typedef std::function< double(const std::vector< double > &, double)> DerivedParamFunc
- typedef std::vector< std::string > StringVec
- typedef std::vector< std::pair< std::string, DerivedParamFunc > > DerivedParamVec

Public Member Functions

- virtual StringVec getParamNames () const

    Gets names of of (independent) model parameters.
- virtual DerivedParamVec getDerivedParams () const

18.6.1 Detailed Description

Base class for all code snippets.

18.6.2 Member Typedef Documentation

18.6.2.1 typedef std::function<double(const std::vector<double> &, double)> Snippet::Base::DerivedParamFunc

18.6.2.2 typedef std::vector<std::pair<std::string, DerivedParamFunc> > Snippet::Base::DerivedParamVec

18.6.2.3 typedef std::vector<std::string> Snippet::Base::StringVec

18.6.3 Member Function Documentation

18.6.3.1 virtual DerivedParamVec Snippet::Base::getDerivedParams ( ) const [inline], [virtual]

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

Reimplemented in NeuronModels::PoissonNew, WeightUpdateModels::PiecewiseSTDP, NewModels::LegacyWrapper< Base, neuronModel, nModels >, NewModels::LegacyWrapper< Base, weightUpdateModel, weightUpdateModels >, NewModels::LegacyWrapper< Base, postSynModel, postSynModels >, NeuronModels::RulkovMap, and PostsynapticModels::ExpCond.

18.6.3.2 virtual StringVec Snippet::Base::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented in NeuronModels::TraubMilesNStep, NeuronModels::TraubMiles, NeuronModels::PoissonNew, NeuronModels::Poisson, WeightUpdateModels::PiecewiseSTDP, NeuronModels::IzhikevichVariable, NewModels::LegacyWrapper< Base, neuronModel, nModels >, NewModels::LegacyWrapper< Base, weightUpdateModel, weightUpdateModels >, NewModels::LegacyWrapper< Base, postSynModel, postSynModels >, NeuronModels::Izhikevich, WeightUpdateModels::StaticGraded, NeuronModels::RulkovMap, InitVarSnippet::Exponential, InitVarSnippet::Normal, PostsynapticModels::DeltaCurr, InitVarSnippet::Uniform, PostsynapticModels::ExpCond, and InitVarSnippet::Constant.

The documentation for this class was generated from the following file:

- snippet.h

18.7 CodeStream::CB Struct Reference

A close bracket marker.

```
#include <codeStream.h>
```

Public Member Functions

- CB (unsigned int level)

Public Attributes

- const unsigned int Level

18.7.1 Detailed Description

A close bracket marker.

Write to code stream `os` using:

```
os << CB(16);
```

18.7.2 Constructor & Destructor Documentation

18.7.2.1 CodeStream::CB::CB (unsigned int *level*) [inline]

18.7.3 Member Data Documentation

18.7.3.1 const unsigned int CodeStream::CB::Level

The documentation for this struct was generated from the following file:

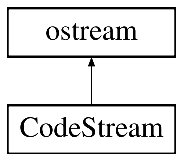
- codeStream.h

18.8 CodeStream Class Reference

Helper class for generating code - automatically inserts brackets, indents etc.

`#include <codeStream.h>`

Inheritance diagram for CodeStream:



Classes

- struct CB

    A close bracket marker.
- struct OB

    An open bracket marker.

Public Member Functions

- CodeStream ()
- CodeStream (std::ostream &stream)
- void setSink (std::ostream &stream)

Friends

- std::ostream & operator<< (std::ostream &s, const OB &ob)
- std::ostream & operator<< (std::ostream &s, const CB &cb)

18.8.1 Detailed Description

Helper class for generating code - automatically inserts brackets, indents etc.

Based heavily on: https://stackoverflow.com/questions/15053753/writing-a-manipulator-for-a-cus

18.8.2 Constructor & Destructor Documentation

18.8.2.1 CodeStream::CodeStream ( ) [inline]

18.8.2.2 CodeStream::CodeStream (std::ostream & *stream*) [inline]

18.8.3 Member Function Documentation

18.8.3.1 void CodeStream::setSink (std::ostream & *stream*) [inline]

18.8.4 Friends And Related Function Documentation

18.8.4.1 std::ostream& operator<< (std::ostream & *s*, const OB & *ob*) [friend]

18.8.4.2 std::ostream& operator<< (std::ostream & *s*, const CB & *cb*) [friend]

The documentation for this class was generated from the following file:

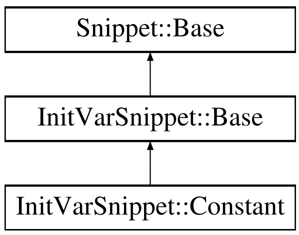
- codeStream.h

18.9 InitVarSnippet::Constant Class Reference

Initialises variable to a constant value.

`#include <initVarSnippet.h>`

Inheritance diagram for InitVarSnippet::Constant:



Public Member Functions

- DECLARE_SNIPPET (InitVarSnippet::Constant, 1)
- SET_CODE ("$(value) = $(constant);")
- virtual StringVec getParamNames () const

    Gets names of of (independent) model parameters.

Additional Inherited Members

18.9.1 Detailed Description

Initialises variable to a constant value.

This snippet takes 1 parameter:

- `value` - The value to intialise the variable to

Note

This snippet type is seldom used directly - NewModels::VarInit has an implicit constructor that, internally, creates one of these snippets

18.9.2 Member Function Documentation

18.9.2.1 InitVarSnippet::Constant::DECLARE_SNIPPET (InitVarSnippet::Constant , 1)

18.9.2.2 virtual StringVec InitVarSnippet::Constant::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

18.9.2.3 InitVarSnippet::Constant::SET_CODE ( )

The documentation for this class was generated from the following file:

- initVarSnippet.h

18.10 CStopWatch Class Reference

Helper class for timing sections of host code in a cross-plarform manner.

`#include <hr_time.h>`

Public Member Functions

- CStopWatch ()
- void startTimer ()

  This method starts the timer.
- void stopTimer ()

  This method stops the timer.
- double getElapsedTime ()

  This method returns the time elapsed between start and stop of the timer in seconds.

18.10.1 Detailed Description

Helper class for timing sections of host code in a cross-plarform manner.

Uses performance counters on windows and microsecond time on Unix

18.10.2 Constructor & Destructor Documentation

18.10.2.1 CStopWatch::CStopWatch ( ) [inline]

18.10.3 Member Function Documentation

18.10.3.1 double CStopWatch::getElapsedTime ( )

This method returns the time elapsed between start and stop of the timer in seconds.

18.10.3.2 void CStopWatch::startTimer ( )

This method starts the timer.

18.10.3.3 void CStopWatch::stopTimer ( )

This method stops the timer.

The documentation for this class was generated from the following files:

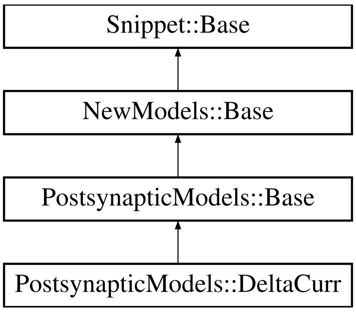
- hr_time.h
- hr_time.cc

18.11 PostsynapticModels::DeltaCurr Class Reference

Simple delta current synapse.

#include <newPostsynapticModels.h>

Inheritance diagram for PostsynapticModels::DeltaCurr:



Public Types

- typedef Snippet::ValueBase< 0 > ParamValues
- typedef NewModels::VarInitContainerBase< 0 > VarValues

Public Member Functions

- virtual std::string getApplyInputCode () const
- virtual StringVec getParamNames () const

  Gets names of of (independent) model parameters.

Static Public Member Functions

- static const DeltaCurr * getInstance ()

18.11.1 Detailed Description

Simple delta current synapse.

Synaptic input provides a direct inject of instantaneous current

18.11.2 Member Typedef Documentation

18.11.2.1 typedef Snippet::ValueBase< 0 > PostsynapticModels::DeltaCurr::ParamValues

18.11.2.2 typedef NewModels::VarInitContainerBase< 0 > PostsynapticModels::DeltaCurr::VarValues

18.11.3 Member Function Documentation

18.11.3.1 virtual std::string PostsynapticModels::DeltaCurr::getApplyInputCode ( ) const [inline], [virtual]

Reimplemented from PostsynapticModels::Base.

18.11.3.2 static const DeltaCurr* PostsynapticModels::DeltaCurr::getInstance ( ) [inline], [static]

18.11.3.3 virtual StringVec PostsynapticModels::DeltaCurr::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

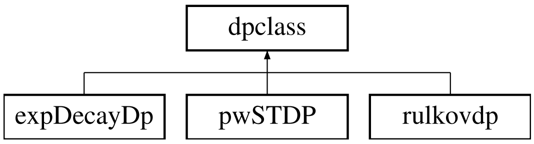
The documentation for this class was generated from the following file:

- newPostsynapticModels.h

18.12 dpclass Class Reference

```
#include <dpclass.h>
```

Inheritance diagram for dpclass:



Public Member Functions

- virtual double calculateDerivedParameter (int, vector< double >, double=0.5)

18.12.1 Member Function Documentation

18.12.1.1 virtual double dpclass::calculateDerivedParameter (int , vector< double > , double = 0.5) [inline], [virtual]

Reimplemented in rulkovdp, pwSTDP, and expDecayDp.

The documentation for this class was generated from the following file:

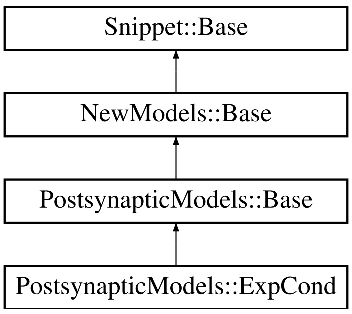
- dpclass.h

18.13 PostsynapticModels::ExpCond Class Reference

Exponential decay with synaptic input treated as a conductance value.

```
#include <newPostsynapticModels.h>
```

Inheritance diagram for PostsynapticModels::ExpCond:



Public Types

- typedef Snippet::ValueBase< 2 > ParamValues
- typedef NewModels::VarInitContainerBase< 0 > VarValues

Public Member Functions

- virtual std::string getDecayCode () const
- virtual std::string getApplyInputCode () const
- virtual StringVec getParamNames () const

    Gets names of of (independent) model parameters.
- virtual DerivedParamVec getDerivedParams () const

Static Public Member Functions

- static const ExpCond * getInstance ()

18.13.1 Detailed Description

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).

18.13.2 Member Typedef Documentation

18.13.2.1 typedef Snippet::ValueBase< 2 > PostsynapticModels::ExpCond::ParamValues

18.13.2.2 typedef NewModels::VarInitContainerBase< 0 > PostsynapticModels::ExpCond::VarValues

18.13.3 Member Function Documentation

18.13.3.1 virtual std::string PostsynapticModels::ExpCond::getApplyInputCode ( ) const [inline], [virtual]

Reimplemented from PostsynapticModels::Base.

18.13.3.2 virtual std::string PostsynapticModels::ExpCond::getDecayCode ( ) const [inline], [virtual]

Reimplemented from PostsynapticModels::Base.

18.13.3.3 virtual DerivedParamVec PostsynapticModels::ExpCond::getDerivedParams ( ) const [inline], [virtual]

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

Reimplemented from Snippet::Base.

18.13.3.4 static const ExpCond* PostsynapticModels::ExpCond::getInstance ( ) [inline], [static]

18.13.3.5 virtual StringVec PostsynapticModels::ExpCond::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

The documentation for this class was generated from the following file:

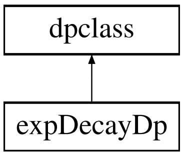
- newPostsynapticModels.h

18.14 expDecayDp Class Reference

Class defining the dependent parameter for exponential decay.

`#include <postSynapseModels.h>`

Inheritance diagram for expDecayDp:



Public Member Functions

- double calculateDerivedParameter (int index, vector< double > pars, double dt=1.0)

18.14.1 Detailed Description

Class defining the dependent parameter for exponential decay.

18.14.2 Member Function Documentation

18.14.2.1 double expDecayDp::calculateDerivedParameter (int *index*, vector< double > *pars*, double *dt* = 1.0) [inline], [virtual]

Reimplemented from dpclass.

The documentation for this class was generated from the following file:

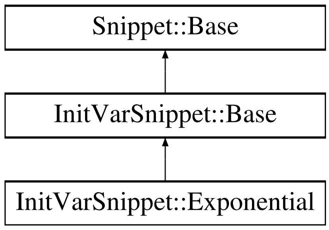
- postSynapseModels.h

18.15 InitVarSnippet::Exponential Class Reference

Initialises variable by sampling from the exponential distribution.

`#include <initVarSnippet.h>`

Inheritance diagram for InitVarSnippet::Exponential:



Public Member Functions

- DECLARE_SNIPPET (InitVarSnippet::Exponential, 1)
- SET_CODE ("$(value) = $(lambda) ∗ $(gennrand_exponential);")
- virtual StringVec getParamNames () const

  Gets names of of (independent) model parameters.

Additional Inherited Members

18.15.1 Detailed Description

Initialises variable by sampling from the exponential distribution.

This snippet takes 1 parameter:

- `lambda` - mean event rate (events per unit time/distance)

18.15.2 Member Function Documentation

18.15.2.1 InitVarSnippet::Exponential::DECLARE_SNIPPET (InitVarSnippet::Exponential , 1)

18.15.2.2 virtual StringVec InitVarSnippet::Exponential::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

18.15.2.3 InitVarSnippet::Exponential::SET_CODE ( )

The documentation for this class was generated from the following file:

- initVarSnippet.h

18.16 FunctionTemplate Struct Reference

`#include <codeGenUtils.h>`

Public Member Functions

- FunctionTemplate operator= (const FunctionTemplate &o)

Public Attributes

- 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.

18.16.1 Detailed Description

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)

18.16.2 Member Function Documentation

18.16.2.1 FunctionTemplate FunctionTemplate::operator= (const FunctionTemplate & *o*) `[inline]`

18.16.3 Member Data Documentation

18.16.3.1 const std::string FunctionTemplate::doublePrecisionTemplate

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

18.16.3.2 const std::string FunctionTemplate::genericName

Generic name used to refer to function in user code.

18.16.3.3 const unsigned int FunctionTemplate::numArguments

Number of function arguments.

18.16.3.4 const std::string FunctionTemplate::singlePrecisionTemplate

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

The documentation for this struct was generated from the following file:

- codeGenUtils.h

18.17 GenericFunction Struct Reference

```
#include <codeGenUtils.h>
```

Public Attributes

- const std::string genericName

  Generic name used to refer to function in user code.
- const unsigned int numArguments

  Number of function arguments.

18.17.1 Detailed Description

Immutable structure for specifying the name and number of arguments of a generic funcion e.g. gennrand_uniform

18.17.2 Member Data Documentation

18.17.2.1 const std::string GenericFunction::genericName

Generic name used to refer to function in user code.

18.17.2.2 const unsigned int GenericFunction::numArguments

Number of function arguments.

The documentation for this struct was generated from the following file:

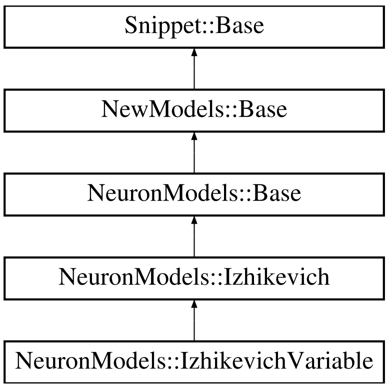
- codeGenUtils.h

18.18 NeuronModels::Izhikevich Class Reference

Izhikevich neuron with fixed parameters [1].

`#include <newNeuronModels.h>`

Inheritance diagram for NeuronModels::Izhikevich:



Public Types

- typedef Snippet::ValueBase< 4 > ParamValues
- typedef NewModels::VarInitContainerBase< 2 > VarValues

Public Member Functions

- virtual std::string getSimCode () const

  Gets the code that defines the execution of one timestep of integration of the neuron model.
- virtual std::string getThresholdConditionCode () const

  Gets code which defines the condition for a true spike in the described neuron model.
- virtual StringVec getParamNames () const

  Gets names of of (independent) model parameters.
- virtual StringPairVec getVars () const

  Gets names and types (as strings) of model variables.

Static Public Member Functions

- static const NeuronModels::Izhikevich * getInstance ()

18.18.1 Detailed Description

Izhikevich neuron with fixed parameters [1].

It is usually described as

$$\begin{aligned} \frac{dV}{dt} &= 0.04V^2 + 5V + 140 - U + I, \\ \frac{dU}{dt} &= a(bV - U), \end{aligned}$$

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

18.18.2 Member Typedef Documentation

18.18.2.1 typedef **Snippet::ValueBase< 4 >** **NeuronModels::Izhikevich::ParamValues**

18.18.2.2 typedef **NewModels::VarInitContainerBase< 2 >** **NeuronModels::Izhikevich::VarValues**

18.18.3 Member Function Documentation

18.18.3.1 static const **NeuronModels::Izhikevich*** NeuronModels::Izhikevich::getInstance ( ) `[inline]`, `[static]`

18.18.3.2 virtual **StringVec** NeuronModels::Izhikevich::getParamNames ( ) const `[inline]`, `[virtual]`

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

Reimplemented in NeuronModels::IzhikevichVariable.

18.18.3.3 virtual std::string NeuronModels::Izhikevich::getSimCode ( ) const `[inline]`, `[virtual]`

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.

Reimplemented from NeuronModels::Base.

18.18.3.4 virtual std::string NeuronModels::Izhikevich::getThresholdConditionCode ( ) const [inline], [virtual]

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

Reimplemented from NeuronModels::Base.

18.18.3.5 virtual StringPairVec NeuronModels::Izhikevich::getVars ( ) const [inline], [virtual]

Gets names and types (as strings) of model variables.

Reimplemented from NewModels::Base.

Reimplemented in NeuronModels::IzhikevichVariable.

The documentation for this class was generated from the following file:

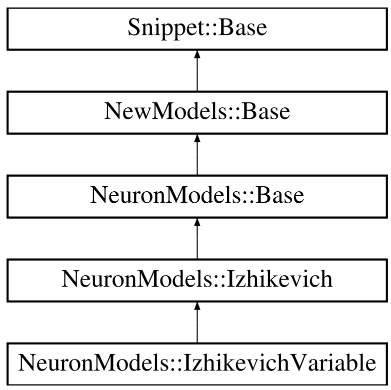
- newNeuronModels.h

18.19 NeuronModels::IzhikevichVariable Class Reference

Izhikevich neuron with variable parameters [1].

```
#include <newNeuronModels.h>
```

Inheritance diagram for NeuronModels::IzhikevichVariable:



Public Types

- typedef Snippet::ValueBase< 0 > ParamValues
- typedef NewModels::VarInitContainerBase< 6 > VarValues

Public Member Functions

- virtual StringVec getParamNames () const

  Gets names of of (independent) model parameters.
- virtual StringPairVec getVars () const

  Gets names and types (as strings) of model variables.

Static Public Member Functions

- static const NeuronModels::IzhikevichVariable ∗ getInstance ()

18.19.1 Detailed Description

Izhikevich neuron with variable parameters [1].

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.

18.19.2 Member Typedef Documentation

18.19.2.1 typedef **Snippet::ValueBase**< 0 > **NeuronModels::IzhikevichVariable::ParamValues**

18.19.2.2 typedef **NewModels::VarInitContainerBase**< 6 > **NeuronModels::IzhikevichVariable::VarValues**

18.19.3 Member Function Documentation

18.19.3.1 static const **NeuronModels::IzhikevichVariable**∗ NeuronModels::IzhikevichVariable::getInstance ( ) `[inline]`, `[static]`

18.19.3.2 virtual **StringVec** NeuronModels::IzhikevichVariable::getParamNames ( ) const `[inline]`, `[virtual]`

Gets names of of (independent) model parameters.

Reimplemented from NeuronModels::Izhikevich.

18.19.3.3 virtual **StringPairVec** NeuronModels::IzhikevichVariable::getVars ( ) const `[inline]`, `[virtual]`

Gets names and types (as strings) of model variables.

Reimplemented from NeuronModels::Izhikevich.

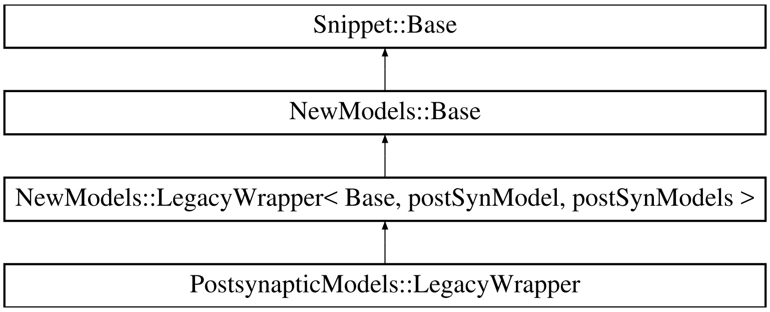
The documentation for this class was generated from the following file:

- newNeuronModels.h

18.20 PostsynapticModels::LegacyWrapper Class Reference

`#include <newPostsynapticModels.h>`

Inheritance diagram for PostsynapticModels::LegacyWrapper:



Public Member Functions

- LegacyWrapper (unsigned int legacyTypeIndex)
- virtual std::string getDecayCode () const
- virtual std::string getApplyInputCode () const
- virtual std::string getSupportCode () const

Additional Inherited Members

18.20.1 Constructor & Destructor Documentation

18.20.1.1 PostsynapticModels::LegacyWrapper::LegacyWrapper (unsigned int *legacyTypeIndex*) [inline]

18.20.2 Member Function Documentation

18.20.2.1 std::string PostsynapticModels::LegacyWrapper::getApplyInputCode ( ) const [virtual]

18.20.2.2 std::string PostsynapticModels::LegacyWrapper::getDecayCode ( ) const [virtual]

18.20.2.3 std::string PostsynapticModels::LegacyWrapper::getSupportCode ( ) const [virtual]

The documentation for this class was generated from the following files:

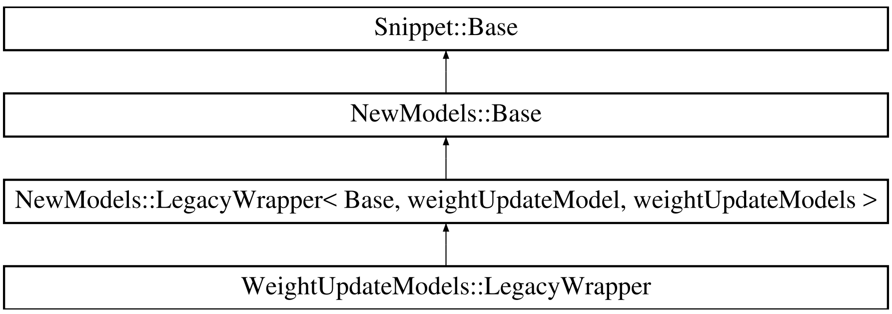
- newPostsynapticModels.h
- newPostsynapticModels.cc

18.21 WeightUpdateModels::LegacyWrapper Class Reference

Wrapper around legacy weight update models stored in weightUpdateModels array of weightUpdateModel objects.

`#include <newWeightUpdateModels.h>`

Inheritance diagram for WeightUpdateModels::LegacyWrapper:



Public Member Functions

- LegacyWrapper (unsigned int legacyTypeIndex)
- 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.
- 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.
- virtual std::string getLearnPostSupportCode () const

  Gets support code to be made available within learnSynapsesPost kernel/function.
- virtual std::string getSynapseDynamicsSuppportCode () const

  Gets support code to be made available within the synapse dynamics kernel/function.
- virtual NewModels::Base::StringPairVec getExtraGlobalParams () const
- virtual bool isPreSpikeTimeRequired () const

  Whether presynaptic spike times are needed or not.
- virtual bool isPostSpikeTimeRequired () const

  Whether postsynaptic spike times are needed or not.

Additional Inherited Members

18.21.1 Detailed Description

Wrapper around legacy weight update models stored in weightUpdateModels array of weightUpdateModel objects.

18.21.2 Constructor & Destructor Documentation

18.21.2.1 WeightUpdateModels::LegacyWrapper::LegacyWrapper (unsigned int *legacyTypeIndex*) [inline]

18.21.3 Member Function Documentation

18.21.3.1 std::string WeightUpdateModels::LegacyWrapper::getEventCode ( ) const [virtual]

Gets code run when events (all the instances where event threshold condition is met) are received.

18.21.3.2 std::string WeightUpdateModels::LegacyWrapper::getEventThresholdConditionCode ( ) const [virtual]

Gets codes to test for events.

18.21.3.3 NewModels::Base::StringPairVec WeightUpdateModels::LegacyWrapper::getExtraGlobalParams ( ) const [virtual]

Gets names and types (as strings) of additional per-population parameters for the weight update model.

18.21.3.4 std::string WeightUpdateModels::LegacyWrapper::getLearnPostCode ( ) const [virtual]

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.

18.21.3.5 std::string WeightUpdateModels::LegacyWrapper::getLearnPostSupportCode ( ) const [virtual]

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.

18.21.3.6 std::string WeightUpdateModels::LegacyWrapper::getSimCode ( ) const [virtual]

Gets simulation code run when 'true' spikes are received.

18.21.3.7 std::string WeightUpdateModels::LegacyWrapper::getSimSupportCode ( ) const [virtual]

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

18.21.3.8 std::string WeightUpdateModels::LegacyWrapper::getSynapseDynamicsCode ( ) const [virtual]

Gets code for synapse dynamics which are independent of spike detection.

18.21.3.9 std::string WeightUpdateModels::LegacyWrapper::getSynapseDynamicsSuppportCode ( ) const [virtual]

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.

18.21.3.10 bool WeightUpdateModels::LegacyWrapper::isPostSpikeTimeRequired ( ) const [virtual]

Whether postsynaptic spike times are needed or not.

18.21.3.11 bool WeightUpdateModels::LegacyWrapper::isPreSpikeTimeRequired ( ) const [virtual]

Whether presynaptic spike times are needed or not.

The documentation for this class was generated from the following files:

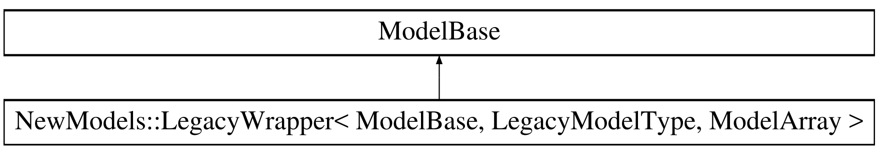
- newWeightUpdateModels.h
- newWeightUpdateModels.cc

18.22 NewModels::LegacyWrapper< ModelBase, LegacyModelType, ModelArray > Class Template Reference

Wrapper around old-style models stored in global arrays and referenced by index.

```
#include <newModels.h>
```

Inheritance diagram for NewModels::LegacyWrapper< ModelBase, LegacyModelType, ModelArray >:



Public Member Functions

- LegacyWrapper (unsigned int legacyTypeIndex)
- virtual StringVec getParamNames () const

    Gets names of of (independent) model parameters.
- virtual DerivedParamVec getDerivedParams () const
- virtual StringPairVec getVars () const

    Gets names and types (as strings) of model variables.

Static Protected Member Functions

- static StringPairVec zipStringVectors (const StringVec &a, const StringVec &b)

Protected Attributes

- const unsigned int m_LegacyTypeIndex

    Index into the array of legacy models.

18.22.1 Detailed Description

template<typename ModelBase, typename LegacyModelType, const std::vector< LegacyModelType > & ModelArray> class NewModels::LegacyWrapper< ModelBase, LegacyModelType, ModelArray >

Wrapper around old-style models stored in global arrays and referenced by index.

18.22.2 Constructor & Destructor Documentation

18.22.2.1 template<typename ModelBase, typename LegacyModelType, const std::vector< LegacyModelType > & ModelArray> NewModels::LegacyWrapper< ModelBase, LegacyModelType, ModelArray >::LegacyWrapper (unsigned int *legacyTypeIndex*) `[inline]`

18.22.3 Member Function Documentation

18.22.3.1 template<typename ModelBase, typename LegacyModelType, const std::vector< LegacyModelType > & ModelArray> virtual DerivedParamVec NewModels::LegacyWrapper< ModelBase, LegacyModelType, ModelArray >::getDerivedParams ( ) const `[inline]`, `[virtual]`

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

18.22.3.2 template<typename ModelBase, typename LegacyModelType, const std::vector< LegacyModelType > & ModelArray> virtual StringVec NewModels::LegacyWrapper< ModelBase, LegacyModelType, ModelArray >::getParamNames ( ) const `[inline]`, `[virtual]`

Gets names of of (independent) model parameters.

18.22.3.3 template<typename ModelBase, typename LegacyModelType, const std::vector< LegacyModelType > & ModelArray> virtual StringPairVec NewModels::LegacyWrapper< ModelBase, LegacyModelType, ModelArray >::getVars ( ) const `[inline]`, `[virtual]`

Gets names and types (as strings) of model variables.

18.22.3.4 template<typename ModelBase, typename LegacyModelType, const std::vector< LegacyModelType > & ModelArray> static StringPairVec NewModels::LegacyWrapper< ModelBase, LegacyModelType, ModelArray >::zipStringVectors (const StringVec & *a*, const StringVec & *b*) [inline], [static], [protected]

18.22.4 Member Data Documentation

18.22.4.1 template<typename ModelBase, typename LegacyModelType, const std::vector< LegacyModelType > & ModelArray> const unsigned int NewModels::LegacyWrapper< ModelBase, LegacyModelType, ModelArray >::m_LegacyTypeIndex [protected]

Index into the array of legacy models.

The documentation for this class was generated from the following file:

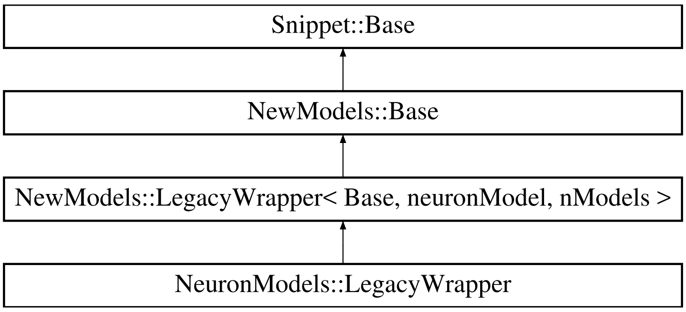
- newModels.h

18.23 NeuronModels::LegacyWrapper Class Reference

Wrapper around legacy weight update models stored in nModels array of neuronModel objects.

```
#include <newNeuronModels.h>
```

Inheritance diagram for NeuronModels::LegacyWrapper:



Public Member Functions

- LegacyWrapper (unsigned int legacyTypeIndex)
- virtual std::string getSimCode () const

    Gets the code that defines the execution of one timestep of integration of the neuron model.
- virtual std::string getThresholdConditionCode () const

    Gets code which defines the condition for a true spike in the described neuron model.
- 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.
- virtual NewModels::Base::StringPairVec getExtraGlobalParams () const
- virtual bool isPoisson () const

Additional Inherited Members

18.23.1 Detailed Description

Wrapper around legacy weight update models stored in nModels array of neuronModel objects.

18.23.2 Constructor & Destructor Documentation

18.23.2.1 NeuronModels::LegacyWrapper::LegacyWrapper (unsigned int *legacyTypeIndex*) [inline]

18.23.3 Member Function Documentation

18.23.3.1 NewModels::Base::StringPairVec NeuronModels::LegacyWrapper::getExtraGlobalParams ( ) const [virtual]

Gets names and types (as strings) of additional per-population parameters for the weight update model.

18.23.3.2 std::string NeuronModels::LegacyWrapper::getResetCode ( ) const [virtual]

Gets code that defines the reset action taken after a spike occurred. This can be empty.

18.23.3.3 std::string NeuronModels::LegacyWrapper::getSimCode ( ) const [virtual]

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.

18.23.3.4 std::string NeuronModels::LegacyWrapper::getSupportCode ( ) const [virtual]

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.

18.23.3.5 std::string NeuronModels::LegacyWrapper::getThresholdConditionCode ( ) const [virtual]

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

18.23.3.6 bool NeuronModels::LegacyWrapper::isPoisson ( ) const [virtual]

The documentation for this class was generated from the following files:

- newNeuronModels.h
- newNeuronModels.cc

18.24 NameIterCtx< Container > Struct Template Reference

```
#include <standardSubstitutions.h>
```

Public Types

- typedef PairKeyConstIter< typename Container::const_iterator > NameIter

Public Member Functions

- NameIterCtx (const Container &c)

Public Attributes

- const Container container

- const NameIter nameBegin
- const NameIter nameEnd

18.24.1 Member Typedef Documentation

18.24.1.1 template< typename Container > typedef PairKeyConstIter< typename Container::const_iterator> NameIterCtx< Container >::NameIter

18.24.2 Constructor & Destructor Documentation

18.24.2.1 template< typename Container > NameIterCtx< Container >::NameIterCtx (const Container & *c*) [inline]

18.24.3 Member Data Documentation

18.24.3.1 template< typename Container > const Container NameIterCtx< Container >::container

18.24.3.2 template< typename Container > const NameIter NameIterCtx< Container >::nameBegin

18.24.3.3 template< typename Container > const NameIter NameIterCtx< Container >::nameEnd

The documentation for this struct was generated from the following file:

- standardSubstitutions.h

18.25 NeuronGroup Class Reference

`#include <neuronGroup.h>`

Public Member Functions

- NeuronGroup (const std::string &name, int numNeurons, const NeuronModels::Base *neuronModel, const std::vector< double > &params, const std::vector< NewModels::VarInit > &varInitialisers)
- NeuronGroup (const NeuronGroup &)=delete
- NeuronGroup ()=delete
- void checkNumDelaySlots (unsigned int requiredDelay)

  Checks delay slots currently provided by the neuron group against a required delay and extends if required.
- void updateVarQueues (const std::string &code)

  Update which variables require queues based on piece of code.
- void setSpikeTimeRequired (bool req)
- void setTrueSpikeRequired (bool req)
- void setSpikeEventRequired (bool req)
- void setSpikeZeroCopyEnabled (bool enabled)

  Function to enable the use of zero-copied memory for spikes (deprecated use NeuronGroup::setSpikeVarMode):
- void setSpikeEventZeroCopyEnabled (bool enabled)

  Function to enable the use of zero-copied memory for spike-like events (deprecated use NeuronGroup::setSpikeEventVarMode):
- void setSpikeTimeZeroCopyEnabled (bool enabled)

  Function to enable the use of zero-copied memory for spike times (deprecated use NeuronGroup::setSpikeTimeVarMode):
- void setVarZeroCopyEnabled (const std::string &varName, bool enabled)

  Function to enable the use zero-copied memory for a particular state variable (deprecated use NeuronGroup::setVarMode):
- void setSpikeVarMode (VarMode mode)

Set variable mode used for variables containing this neuron group's output spikes.
- void setSpikeEventVarMode (VarMode mode)

  Set variable mode used for variables containing this neuron group's output spike events.
- void setSpikeTimeVarMode (VarMode mode)

  Set variable mode used for variables containing this neuron group's output spike times.
- void setVarMode (const std::string &varName, VarMode mode)

  Set variable mode of neuron model state variable.
- void setClusterIndex (int hostID, int deviceID)
- void addSpkEventCondition (const std::string &code, const std::string &supportCodeNamespace)
- void addInSyn (SynapseGroup ∗synapseGroup)
- void addOutSyn (SynapseGroup ∗synapseGroup)
- void initDerivedParams (double dt)
- void calcSizes (unsigned int blockSize, unsigned int &idStart, unsigned int &paddedIDStart)
- const std::string & getName () const
- unsigned int getNumNeurons () const

  Gets number of neurons in group.
- const std::pair< unsigned int, unsigned int > & getPaddedIDRange () const
- const std::pair< unsigned int, unsigned int > & getIDRange () const
- const NeuronModels::Base ∗ getNeuronModel () const

  Gets the neuron model used by this group.
- const std::vector< double > & getParams () const
- const std::vector< double > & getDerivedParams () const
- const std::vector< NewModels::VarInit > & getVarInitialisers () const
- const std::vector< SynapseGroup ∗ > & getInSyn () const

  Gets pointers to all synapse groups which provide input to this neuron group.
- const std::vector< SynapseGroup ∗ > & getOutSyn () const

  Gets pointers to all synapse groups emanating from this neuron group.
- bool isSpikeTimeRequired () const
- bool isTrueSpikeRequired () const
- bool isSpikeEventRequired () const
- bool isQueueRequired () const
- bool isVarQueueRequired (const std::string &var) const
- bool isVarQueueRequired (size_t index) const
- bool isVarQueueRequired () const
- const std::set< std::pair< std::string, std::string > > & getSpikeEventCondition () const
- unsigned int getNumDelaySlots () const
- bool isDelayRequired () const
- bool isSpikeZeroCopyEnabled () const
- bool isSpikeEventZeroCopyEnabled () const
- bool isSpikeTimeZeroCopyEnabled () const
- bool isZeroCopyEnabled () const
- bool isVarZeroCopyEnabled (const std::string &var) const
- VarMode getSpikeVarMode () const

  Get variable mode used for variables containing this neuron group's output spikes.
- VarMode getSpikeEventVarMode () const

  Get variable mode used for variables containing this neuron group's output spike events.
- VarMode getSpikeTimeVarMode () const

  Get variable mode used for variables containing this neuron group's output spike times.
- VarMode getVarMode (const std::string &varName) const

  Get variable mode used by neuron model state variable.
- VarMode getVarMode (size_t index) const

  Get variable mode used by neuron model state variable.
- bool isParamRequiredBySpikeEventCondition (const std::string &pnamefull) const

Do any of the spike event conditions tested by this neuron require specified parameter.
- void addExtraGlobalParams (std::map< std::string, std::string > &kernelParameters) const
- bool isInitCodeRequired () const

  Does this neuron group require any initialisation code to be run.
- bool isSimRNGRequired () const

  Does this neuron group require an RNG to simulate.
- bool isInitRNGRequired (VarInit varInitMode) const

  Does this neuron group require an RNG for it's init code.
- bool isDeviceVarInitRequired () const

  Is device var init code required for any variables in this neuron group.
- bool canRunOnCPU () const

  Can this neuron group run on the CPU?
- std::string getQueueOffset (const std::string &devPrefix) const

18.25.1 Constructor & Destructor Documentation

18.25.1.1 NeuronGroup::NeuronGroup (const std::string & *name*, int *numNeurons*, const **NeuronModels::Base** ∗ *neuronModel*, const std::vector< double > & *params*, const std::vector< **NewModels::VarInit** > & *varInitialisers*) [inline]

18.25.1.2 NeuronGroup::NeuronGroup (const **NeuronGroup** &) [delete]

18.25.1.3 NeuronGroup::NeuronGroup ( ) [delete]

18.25.2 Member Function Documentation

18.25.2.1 void NeuronGroup::addExtraGlobalParams (std::map< std::string, std::string > & *kernelParameters*) const

18.25.2.2 void NeuronGroup::addInSyn (**SynapseGroup** ∗ *synapseGroup*) [inline]

18.25.2.3 void NeuronGroup::addOutSyn (**SynapseGroup** ∗ *synapseGroup*) [inline]

18.25.2.4 void NeuronGroup::addSpkEventCondition (const std::string & *code*, const std::string & *supportCodeNamespace*)

18.25.2.5 void NeuronGroup::calcSizes (unsigned int *blockSize*, unsigned int & *idStart*, unsigned int & *paddedIDStart*)

18.25.2.6 bool NeuronGroup::canRunOnCPU ( ) const

Can this neuron group run on the CPU?

If we are running in CPU_ONLY mode this is always true, but some GPU functionality will prevent models being run on both CPU and GPU.

18.25.2.7 void NeuronGroup::checkNumDelaySlots (unsigned int *requiredDelay*)

Checks delay slots currently provided by the neuron group against a required delay and extends if required.

18.25.2.8 const std::vector<double>& NeuronGroup::getDerivedParams ( ) const [inline]

18.25.2.9 const std::pair<unsigned int, unsigned int>& NeuronGroup::getIDRange ( ) const [inline]

18.25.2.10 const std::vector<**SynapseGroup**∗>& NeuronGroup::getInSyn ( ) const [inline]

Gets pointers to all synapse groups which provide input to this neuron group.

18.25.2.11 const std::string& NeuronGroup::getName ( ) const [inline]

18.25.2.12 const **NeuronModels::Base**∗ NeuronGroup::getNeuronModel ( ) const [inline]

Gets the neuron model used by this group.

18.25.2.13 unsigned int NeuronGroup::getNumDelaySlots ( ) const [inline]

18.25.2.14 unsigned int NeuronGroup::getNumNeurons ( ) const [inline]

Gets number of neurons in group.

18.25.2.15 const std::vector<SynapseGroup*>& NeuronGroup::getOutSyn ( ) const [inline]

Gets pointers to all synapse groups emanating from this neuron group.

18.25.2.16 const std::pair<unsigned int, unsigned int>& NeuronGroup::getPaddedIDRange ( ) const [inline]

18.25.2.17 const std::vector<double>& NeuronGroup::getParams ( ) const [inline]

18.25.2.18 std::string NeuronGroup::getQueueOffset (const std::string & *devPrefix*) const

18.25.2.19 const std::set<std::pair<std::string, std::string> >& NeuronGroup::getSpikeEventCondition ( ) const [inline]

18.25.2.20 VarMode NeuronGroup::getSpikeEventVarMode ( ) const [inline]

Get variable mode used for variables containing this neuron group's output spike events.

18.25.2.21 VarMode NeuronGroup::getSpikeTimeVarMode ( ) const [inline]

Get variable mode used for variables containing this neuron group's output spike times.

18.25.2.22 VarMode NeuronGroup::getSpikeVarMode ( ) const [inline]

Get variable mode used for variables containing this neuron group's output spikes.

18.25.2.23 const std::vector<NewModels::VarInit>& NeuronGroup::getVarInitialisers ( ) const [inline]

18.25.2.24 VarMode NeuronGroup::getVarMode (const std::string & *varName*) const

Get variable mode used by neuron model state variable.

18.25.2.25 VarMode NeuronGroup::getVarMode (size_t *index*) const [inline]

Get variable mode used by neuron model state variable.

18.25.2.26 void NeuronGroup::initDerivedParams (double *dt*)

18.25.2.27 bool NeuronGroup::isDelayRequired ( ) const [inline]

18.25.2.28 bool NeuronGroup::isDeviceVarInitRequired ( ) const

Is device var init code required for any variables in this neuron group.

18.25.2.29 bool NeuronGroup::isInitCodeRequired ( ) const

Does this neuron group require any initialisation code to be run.

18.25.2.30 bool NeuronGroup::isInitRNGRequired (VarInit *varInitMode*) const

Does this neuron group require an RNG for it's init code.

18.25.2.31 bool NeuronGroup::isParamRequiredBySpikeEventCondition (const std::string & *pnamefull*) const

Do any of the spike event conditions tested by this neuron require specified parameter.

18.25.2.32 bool NeuronGroup::isQueueRequired ( ) const [inline]

18.25.2.33 bool NeuronGroup::isSimRNGRequired ( ) const

Does this neuron group require an RNG to simulate.

18.25.2.34 bool NeuronGroup::isSpikeEventRequired ( ) const [inline]

18.25.2.35 bool NeuronGroup::isSpikeEventZeroCopyEnabled ( ) const [inline]

18.25.2.36 bool NeuronGroup::isSpikeTimeRequired ( ) const [inline]

18.25.2.37 bool NeuronGroup::isSpikeTimeZeroCopyEnabled ( ) const [inline]

18.25.2.38 bool NeuronGroup::isSpikeZeroCopyEnabled ( ) const [inline]

18.25.2.39 bool NeuronGroup::isTrueSpikeRequired ( ) const [inline]

18.25.2.40 bool NeuronGroup::isVarQueueRequired (const std::string & *var*) const

18.25.2.41 bool NeuronGroup::isVarQueueRequired (size_t *index*) const [inline]

18.25.2.42 bool NeuronGroup::isVarQueueRequired ( ) const [inline]

18.25.2.43 bool NeuronGroup::isVarZeroCopyEnabled (const std::string & *var*) const [inline]

18.25.2.44 bool NeuronGroup::isZeroCopyEnabled ( ) const

18.25.2.45 void NeuronGroup::setClusterIndex (int *hostID,* int *deviceID*) [inline]

18.25.2.46 void NeuronGroup::setSpikeEventRequired (bool *req*) [inline]

18.25.2.47 void NeuronGroup::setSpikeEventVarMode (VarMode *mode*) [inline]

Set variable mode used for variables containing this neuron group's output spike events.

This is ignored for CPU simulations

18.25.2.48 void NeuronGroup::setSpikeEventZeroCopyEnabled (bool *enabled*) [inline]

Function to enable the use of zero-copied memory for spike-like events (deprecated use NeuronGroup::setSpike↩EventVarMode):

May improve IO performance at the expense of kernel performance

18.25.2.49 void NeuronGroup::setSpikeTimeRequired (bool *req*) [inline]

18.25.2.50 void NeuronGroup::setSpikeTimeVarMode (VarMode *mode*) [inline]

Set variable mode used for variables containing this neuron group's output spike times.

This is ignored for CPU simulations

18.25.2.51 void NeuronGroup::setSpikeTimeZeroCopyEnabled (bool *enabled*) [inline]

Function to enable the use of zero-copied memory for spike times (deprecated use NeuronGroup::setSpikeTime↩VarMode):

May improve IO performance at the expense of kernel performance

18.25.2.52 void NeuronGroup::setSpikeVarMode (VarMode *mode*) [inline]

Set variable mode used for variables containing this neuron group's output spikes.

This is ignored for CPU simulations

18.25.2.53 void NeuronGroup::setSpikeZeroCopyEnabled (bool *enabled*) [inline]

Function to enable the use of zero-copied memory for spikes (deprecated use NeuronGroup::setSpikeVarMode):

May improve IO performance at the expense of kernel performance

18.25.2.54 void NeuronGroup::setTrueSpikeRequired (bool *req*) [inline]

18.25.2.55 void NeuronGroup::setVarMode (const std::string & *varName,* VarMode *mode*)

Set variable mode of neuron model state variable.

This is ignored for CPU simulations

18.25.2.56 void NeuronGroup::setVarZeroCopyEnabled (const std::string & *varName,* bool *enabled*) [inline]

Function to enable the use zero-copied memory for a particular state variable (deprecated use NeuronGroup::set←VarMode):

May improve IO performance at the expense of kernel performance

18.25.2.57 void NeuronGroup::updateVarQueues (const std::string & *code*)

Update which variables require queues based on piece of code.

The documentation for this class was generated from the following files:

- neuronGroup.h
- neuronGroup.cc

18.26 neuronModel Class Reference

class for specifying a neuron model.

`#include <neuronModels.h>`

Public Member Functions

- neuronModel ()

  Constructor for neuronModel objects.
- ∼neuronModel ()

  Destructor for neuronModel objects.

Public Attributes

- string simCode

  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.
- string thresholdConditionCode

  Code evaluating to a bool (e.g. "V > 20") that defines the condition for a true spike in the described neuron model.
- string resetCode

  Code that defines the reset action taken after a spike occurred. This can be empty.
- string supportCode

  Support code is made available within the neuron kernel definition file and is meant 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.
- vector< string > varNames

Names of the variables in the neuron model.

- vector< string > tmpVarNames

  never used
- vector< string > varTypes

  Types of the variable named above, e.g. "float". Names and types are matched by their order of occurrence in the vector.
- vector< string > tmpVarTypes

  never used
- vector< string > pNames

  Names of (independent) parameters of the model.
- vector< string > dpNames

  Names of dependent parameters of the model. The dependent parameters are functions of independent parameters that enter into the neuron model. To avoid unecessary computational overhead, these parameters are calculated at compile time and inserted as explicit values into the generated code. See method NNmodel::initDerivedNeuronPara for how this is done.
- vector< string > extraGlobalNeuronKernelParameters

  Additional parameter in the neuron kernel; it is translated to a population specific name but otherwise assumed to be one parameter per population rather than per neuron.
- vector< string > extraGlobalNeuronKernelParameterTypes

  Additional parameters in the neuron kernel; they are translated to a population specific name but otherwise assumed to be one parameter per population rather than per neuron.
- dpclass * dps

  Derived parameters.

18.26.1 Detailed Description

class for specifying a neuron model.

18.26.2 Constructor & Destructor Documentation

18.26.2.1 neuronModel::neuronModel ( )

Constructor for neuronModel objects.

18.26.2.2 neuronModel::∼neuronModel ( )

Destructor for neuronModel objects.

18.26.3 Member Data Documentation

18.26.3.1 vector<string> neuronModel::dpNames

Names of dependent parameters of the model. The dependent parameters are functions of independent parameters that enter into the neuron model. To avoid unecessary computational overhead, these parameters are calculated at compile time and inserted as explicit values into the generated code. See method NNmodel::initDerivedNeuronPara for how this is done.

18.26.3.2 dpclass* neuronModel::dps

Derived parameters.

18.26.3.3 vector<string> neuronModel::extraGlobalNeuronKernelParameters

Additional parameter in the neuron kernel; it is translated to a population specific name but otherwise assumed to be one parameter per population rather than per neuron.

18.26.3.4 vector<string> neuronModel::extraGlobalNeuronKernelParameterTypes

Additional parameters in the neuron kernel; they are translated to a population specific name but otherwise assumed to be one parameter per population rather than per neuron.

18.26.3.5 vector<string> neuronModel::pNames

Names of (independent) parameters of the model.

18.26.3.6 string neuronModel::resetCode

Code that defines the reset action taken after a spike occurred. This can be empty.

18.26.3.7 string neuronModel::simCode

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.

18.26.3.8 string neuronModel::supportCode

Support code is made available within the neuron kernel definition file and is meant 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.

18.26.3.9 string neuronModel::thresholdConditionCode

Code evaluating to a bool (e.g. "V > 20") that defines the condition for a true spike in the described neuron model.

18.26.3.10 vector<string> neuronModel::tmpVarNames

never used

18.26.3.11 vector<string> neuronModel::tmpVarTypes

never used

18.26.3.12 vector<string> neuronModel::varNames

Names of the variables in the neuron model.

18.26.3.13 vector<string> neuronModel::varTypes

Types of the variable named above, e.g. "float". Names and types are matched by their order of occurrence in the vector.

The documentation for this class was generated from the following files:

- neuronModels.h
- neuronModels.cc

18.27 NNmodel Class Reference

```
#include <modelSpec.h>
```

Public Types

- typedef map< string, NeuronGroup >::value_type NeuronGroupValueType
- typedef map< string, SynapseGroup >::value_type SynapseGroupValueType

Public Member Functions

- NNmodel ()
- ∼NNmodel ()
- void setName (const std::string &)

  Method to set the neuronal network model name.
- void setPrecision (FloatType)

  Set numerical precision for floating point.
- void setDT (double)

  Set the integration step size of the model.
- void setTiming (bool)

  Set whether timers and timing commands are to be included.
- void setSeed (unsigned int)

  Set the random seed (disables automatic seeding if argument not 0).
- void setRNType (const std::string &type)

  Sets the underlying type for random number generation (default: uint64_t)
- void setGPUDevice (int)

  Sets the underlying type for random number generation (default: uint64_t)
- string scalarExpr (const double) const

  Get the string literal that should be used to represent a value in the model's floating-point type.
- void setPopulationSums ()

  Set the accumulated sums of lowest multiple of kernel block size >= group sizes for all simulated groups.
- void finalize ()

  Declare that the model specification is finalised in modelDefinition().
- bool zeroCopyInUse () const

  Are any variables in any populations in this model using zero-copy memory?
- bool isDeviceInitRequired () const

  Does this model require device initialisation kernel.
- bool isDeviceSparseInitRequired () const

  Does this model require a device sparse initialisation kernel.
- bool isHostRNGRequired () const

  Do any populations or initialisation code in this model require a host RNG?
- bool isDeviceRNGRequired () const

  Do any populations or initialisation code in this model require a device RNG?
- bool canRunOnCPU () const

  Can this model run on the CPU?
- const std::string & getName () const

  Gets the name of the neuronal network model.
- const std::string & getPrecision () const

  Gets the floating point numerical precision.
- unsigned int getResetKernel () const

  Which kernel should contain the reset logic? Specified in terms of GENN_FLAGS.
- double getDT () const

  Gets the model integration step size.
- unsigned int getSeed () const

  Get the random seed.
- const std::string & getRNType () const

  Gets the underlying type for random number generation (default: uint64_t)
- bool isFinalized () const

  Is the model specification finalized.
- bool isTimingEnabled () const

Are timers and timing commands enabled.

- const map< string, NeuronGroup > & getNeuronGroups () const

  Get std::map containing all named NeuronGroup objects in model.
- const map< string, string > & getNeuronKernelParameters () const

  Gets std::map containing names and types of each parameter that should be passed through to the neuron kernel.
- unsigned int getNeuronGridSize () const

  Gets the size of the neuron kernel thread grid.
- unsigned int getNumNeurons () const

  How many neurons make up the entire model.
- const NeuronGroup ∗ findNeuronGroup (const std::string &name) const

  Find a neuron group by name.
- NeuronGroup ∗ findNeuronGroup (const std::string &name)

  Find a neuron group by name.
- NeuronGroup ∗ addNeuronPopulation (const string &, unsigned int, unsigned int, const double ∗, const double ∗)

  Method for adding a neuron population to a neuronal network model, using C++ string for the name of the population.
- NeuronGroup ∗ addNeuronPopulation (const string &, unsigned int, unsigned int, const vector< double > &, const vector< double > &)

  Method for adding a neuron population to a neuronal network model, using C++ string for the name of the population.
- template<typename NeuronModel >
  NeuronGroup ∗ addNeuronPopulation (const string &name, unsigned int size, const NeuronModel ∗model, const typename NeuronModel::ParamValues &paramValues, const typename NeuronModel::VarValues &varInitialisers)

  Adds a new neuron group to the model using a neuron model managed by the user.
- template<typename NeuronModel >
  NeuronGroup ∗ addNeuronPopulation (const string &name, unsigned int size, const typename NeuronModel::ParamValues &paramValues, const typename NeuronModel::VarValues &varInitialisers)

  Adds a new neuron group to the model using a singleton neuron model created using standard DECLARE_MODEL and IMPLEMENT_MODEL macros.
- void setNeuronClusterIndex (const string &neuronGroup, int hostID, int deviceID)

  Function for setting which host and which device a neuron group will be simulated on.
- void activateDirectInput (const string &, unsigned int type)

  This function defines the type of the explicit input to the neuron model. Current options are common constant input to all neurons, input from a file and input defines as a rule.
- void setConstInp (const string &, double)

  This function has been deprecated in GeNN 2.2.
- const map< string, SynapseGroup > & getSynapseGroups () const

  Get std::map containing all named SynapseGroup objects in model.
- const map< string, std::pair< unsigned int, unsigned int > > & getSynapsePostLearnGroups () const
- const map< string, std::pair< unsigned int, unsigned int > > & getSynapseDynamicsGroups () const
- const map< string, string > & getSynapseKernelParameters () const

  Gets std::map containing names and types of each parameter that should be passed through to the synapse kernel.
- const map< string, string > & getSimLearnPostKernelParameters () const

  Gets std::map containing names and types of each parameter that should be passed through to the postsynaptic learning kernel.
- const map< string, string > & getSynapseDynamicsKernelParameters () const

  Gets std::map containing names and types of each parameter that should be passed through to the synapse dynamics kernel.
- unsigned int getSynapseKernelGridSize () const

  Gets the size of the synapse kernel thread grid.
- unsigned int getSynapsePostLearnGridSize () const

  Gets the size of the post-synaptic learning kernel thread grid.
- unsigned int getSynapseDynamicsGridSize () const

Gets the size of the synapse dynamics kernel thread grid.

- const SynapseGroup ∗ findSynapseGroup (const std::string &name) const

  Find a synapse group by name.

- SynapseGroup ∗ findSynapseGroup (const std::string &name)

  Find a synapse group by name.

- bool isSynapseGroupDynamicsRequired (const std::string &name) const

  Does named synapse group have synapse dynamics.

- bool isSynapseGroupPostLearningRequired (const std::string &name) const

  Does named synapse group have post-synaptic learning.

- SynapseGroup ∗ addSynapsePopulation (const string &name, unsigned int syntype, SynapseConnType conntype, SynapseGType gtype, const string &src, const string &trg, const double ∗p)

  This function has been depreciated as of GeNN 2.2.

- SynapseGroup ∗ addSynapsePopulation (const string &, unsigned int, SynapseConnType, SynapseGType, unsigned int, unsigned int, const string &, const string &, const double ∗, const double ∗, const double ∗)

  Overloaded version without initial variables for synapses.

- SynapseGroup ∗ addSynapsePopulation (const string &, unsigned int, SynapseConnType, SynapseGType, unsigned int, unsigned int, const string &, const string &, const double ∗, const double ∗, const double ∗, const double ∗)

  Method for adding a synapse population to a neuronal network model, using C++ string for the name of the population.

- SynapseGroup ∗ addSynapsePopulation (const string &, unsigned int, SynapseConnType, SynapseGType, unsigned int, unsigned int, const string &, const string &, const vector< double > &, const vector< double > &, const vector< double > &, const vector< double > &)

  Method for adding a synapse population to a neuronal network model, using C++ string for the name of the population.

- template<typename WeightUpdateModel , typename PostsynapticModel >
  SynapseGroup ∗ addSynapsePopulation (const string &name, SynapseMatrixType mtype, unsigned int delaySteps, const string &src, const string &trg, const WeightUpdateModel ∗wum, const typename WeightUpdateModel::ParamValues &weightParamValues, const typename WeightUpdateModel::VarValues &weightVarInitialisers, const PostsynapticModel ∗psm, const typename PostsynapticModel::ParamValues &postsynapticParamValues, const typename PostsynapticModel::VarValues &postsynapticVarInitialisers)

  Adds a synapse population to the model using weight update and postsynaptic models managed by the user.

- template<typename WeightUpdateModel , typename PostsynapticModel >
  SynapseGroup ∗ addSynapsePopulation (const string &name, SynapseMatrixType mtype, unsigned int delaySteps, const string &src, const string &trg, const typename WeightUpdateModel::ParamValues &weightParamValues, const typename WeightUpdateModel::VarValues &weightVarInitialisers, const typename PostsynapticModel::ParamValues &postsynapticParamValues, const typename PostsynapticModel::↩ VarValues &postsynapticVarInitialisers)

  Adds a synapse population to the model using singleton weight update and postsynaptic models created using standard DECLARE_MODEL and IMPLEMENT_MODEL macros.

- void setSynapseG (const string &, double)

  This function has been depreciated as of GeNN 2.2.

- void setMaxConn (const string &, unsigned int)

  This function defines the maximum number of connections for a neuron in the population.

- void setSpanTypeToPre (const string &)

  Method for switching the execution order of synapses to pre-to-post.

- void setSynapseClusterIndex (const string &synapseGroup, int hostID, int deviceID)

  Function for setting which host and which device a synapse group will be simulated on.

18.27.1 Member Typedef Documentation

18.27.1.1 typedef map<string, NeuronGroup>::value_type NNmodel::NeuronGroupValueType

18.27.1.2 typedef map<string, SynapseGroup>::value_type NNmodel::SynapseGroupValueType

18.27.2 Constructor & Destructor Documentation

18.27.2.1 NNmodel::NNmodel ( )

18.27.2.2 NNmodel::∼NNmodel ( )

18.27.3 Member Function Documentation

18.27.3.1 void NNmodel::activateDirectInput (const string & , unsigned int *type*)

This function defines the type of the explicit input to the neuron model. Current options are common constant input to all neurons, input from a file and input defines as a rule.

Parameters

type	Type of input: 1 if common input, 2 if custom input from file, 3 if custom input as a rule

18.27.3.2 NeuronGroup ∗ NNmodel::addNeuronPopulation (const string & *name*, unsigned int *nNo*, unsigned int *type*, const double ∗ *p*, const double ∗ *ini*)

Method for adding a neuron population to a neuronal network model, using C++ string for the name of the population.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This function adds a neuron population to a neuronal network models, assigning the name, the number of neurons in the group, the neuron type, parameters and initial values, the latter two defined as double ∗

Parameters

name	The name of the neuron population
nNo	Number of neurons in the population
type	Type of the neurons, refers to either a standard type or user-defined type
p	Parameters of this neuron type
ini	Initial values for variables of this neuron type

18.27.3.3 NeuronGroup ∗ NNmodel::addNeuronPopulation (const string & *name*, unsigned int *nNo*, unsigned int *type*, const vector< double > & *p*, const vector< double > & *ini*)

Method for adding a neuron population to a neuronal network model, using C++ string for the name of the population.

This function adds a neuron population to a neuronal network models, assigning the name, the number of neurons in the group, the neuron type, parameters and initial values. The latter two defined as STL vectors of double.

Parameters

name	The name of the neuron population
nNo	Number of neurons in the population
type	Type of the neurons, refers to either a standard type or user-defined type
p	Parameters of this neuron type
ini	Initial values for variables of this neuron type

18.27.3.4 template<typename NeuronModel > NeuronGroup∗ NNmodel::addNeuronPopulation (const string & *name,* unsigned int *size,* const NeuronModel ∗ *model,* const typename NeuronModel::ParamValues & *paramValues,* const typename NeuronModel::VarValues & *varInitialisers*) [inline]

Adds a new neuron group to the model using a neuron model managed by the user.

Template Parameters

NeuronModel	type of neuron model (derived from NeuronModels::Base).

Parameters

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.

Returns

pointer to newly created NeuronGroup

18.27.3.5 template<typename NeuronModel > NeuronGroup∗ NNmodel::addNeuronPopulation (const string & *name,* unsigned int *size,* const typename NeuronModel::ParamValues & *paramValues,* const typename NeuronModel::VarValues & *varInitialisers*) [inline]

Adds a new neuron group to the model using a singleton neuron model created using standard DECLARE_MODEL and IMPLEMENT_MODEL macros.

Template Parameters

NeuronModel	type of neuron model (derived from NeuronModels::Base).

Parameters

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.

Returns

pointer to newly created NeuronGroup

18.27.3.6 SynapseGroup ∗ NNmodel::addSynapsePopulation (const string & *name,* unsigned int *syntype,* SynapseConnType *conntype,* SynapseGType *gtype,* const string & *src,* const string & *trg,* const double ∗ *p*)

This function has been depreciated as of GeNN 2.2.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This deprecated function is provided for compatibility with the previous release of GeNN. Default values are provide for new parameters, it is strongly recommended these be selected explicity via the new version othe function

Parameters

name	The name of the synapse population
syntype	The type of synapse to be added (i.e. learning mode)
conntype	The type of synaptic connectivity
gtype	The way how the synaptic conductivity g will be defined
src	Name of the (existing!) pre-synaptic neuron population
trg	Name of the (existing!) post-synaptic neuron population
p	A C-type array of doubles that contains synapse parameter values (common to all synapses of the population) which will be used for the defined synapses.

18.27.3.7 SynapseGroup ∗ NNmodel::addSynapsePopulation (const string & *name*, unsigned int *syntype*, SynapseConnType *conntype*, SynapseGType *gtype*, unsigned int *delaySteps*, unsigned int *postsyn*, const string & *src*, const string & *trg*, const double ∗ *p*, const double ∗ *PSVini*, const double ∗ *ps*)

Overloaded version without initial variables for synapses.

Overloaded old version (deprecated)

Parameters

name	The name of the synapse population
syntype	The type of synapse to be added (i.e. learning mode)
conntype	The type of synaptic connectivity
gtype	The way how the synaptic conductivity g will be defined
delaySteps	Number of delay slots
postsyn	Postsynaptic integration method
src	Name of the (existing!) pre-synaptic neuron population
trg	Name of the (existing!) post-synaptic neuron population
p	A C-type array of doubles that contains synapse parameter values (common to all synapses of the population) which will be used for the defined synapses.
PSVini	A C-type array of doubles that contains the initial values for postsynaptic mechanism variables (common to all synapses of the population) which will be used for the defined synapses.
ps	A C-type array of doubles that contains postsynaptic mechanism parameter values (common to all synapses of the population) which will be used for the defined synapses.

18.27.3.8 SynapseGroup ∗ NNmodel::addSynapsePopulation (const string & *name*, unsigned int *syntype*, SynapseConnType *conntype*, SynapseGType *gtype*, unsigned int *delaySteps*, unsigned int *postsyn*, const string & *src*, const string & *trg*, const double ∗ *synini*, const double ∗ *p*, const double ∗ *PSVini*, const double ∗ *ps*)

Method for adding a synapse population to a neuronal network model, using C++ string for the name of the population.

This function adds a synapse population to a neuronal network model, assigning the name, the synapse type, the connectivity type, the type of conductance specification, the source and destination neuron populations, and the synaptic parameters.

Parameters

name	The name of the synapse population
syntype	The type of synapse to be added (i.e. learning mode)
conntype	The type of synaptic connectivity
gtype	The way how the synaptic conductivity g will be defined

Parameters

delaySteps	Number of delay slots
postsyn	Postsynaptic integration method
src	Name of the (existing!) pre-synaptic neuron population
trg	Name of the (existing!) post-synaptic neuron population
synini	A C-type array of doubles that contains the initial values for synapse variables (common to all synapses of the population) which will be used for the defined synapses.
p	A C-type array of doubles that contains synapse parameter values (common to all synapses of the population) which will be used for the defined synapses.
PSVini	A C-type array of doubles that contains the initial values for postsynaptic mechanism variables (common to all synapses of the population) which will be used for the defined synapses.
ps	A C-type array of doubles that contains postsynaptic mechanism parameter values (common to all synapses of the population) which will be used for the defined synapses.

18.27.3.9 SynapseGroup ∗ NNmodel::addSynapsePopulation (const string & *name*, unsigned int *syntype*, SynapseConnType *conntype*, SynapseGType *gtype*, unsigned int *delaySteps*, unsigned int *postsyn*, const string & *src*, const string & *trg*, const vector< double > & *synini*, const vector< double > & *p*, const vector< double > & *PSVini*, const vector< double > & *ps*)

Method for adding a synapse population to a neuronal network model, using C++ string for the name of the population.

This function adds a synapse population to a neuronal network model, assigning the name, the synapse type, the connectivity type, the type of conductance specification, the source and destination neuron populations, and the synaptic parameters.

Parameters

name	The name of the synapse population
syntype	The type of synapse to be added (i.e. learning mode)
conntype	The type of synaptic connectivity
gtype	The way how the synaptic conductivity g will be defined
delaySteps	Number of delay slots
postsyn	Postsynaptic integration method
src	Name of the (existing!) pre-synaptic neuron population
trg	Name of the (existing!) post-synaptic neuron population
synini	A C-type array of doubles that contains the initial values for synapse variables (common to all synapses of the population) which will be used for the defined synapses.
p	A C-type array of doubles that contains synapse parameter values (common to all synapses of the population) which will be used for the defined synapses.
PSVini	A C-type array of doubles that contains the initial values for postsynaptic mechanism variables (common to all synapses of the population) which will be used for the defined synapses.
ps	A C-type array of doubles that contains postsynaptic mechanism parameter values (common to all synapses of the population) which will be used for the defined synapses.

18.27.3.10 template<typename WeightUpdateModel , typename PostsynapticModel > SynapseGroup* NNmodel::addSynapsePopulation (const string & *name*, SynapseMatrixType *mtype*, unsigned int *delaySteps*, const string & *src*, const string & *trg*, const WeightUpdateModel * *wum*, const typename WeightUpdateModel::ParamValues & *weightParamValues*, const typename WeightUpdateModel::VarValues & *weightVarInitialisers*, const PostsynapticModel * *psm*, const typename PostsynapticModel::ParamValues & *postsynapticParamValues*, const typename PostsynapticModel::VarValues & *postsynapticVarInitialisers*) [inline]

Adds a synapse population to the model using weight update and postsynaptic models managed by the user.

Template Parameters

WeightUpdateModel	type of weight update model (derived from WeightUpdateModels::Base).
PostsynapticModel	type of postsynaptic model (derived from PostsynapticModels::Base).

Parameters

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.
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.

Returns

pointer to newly created SynapseGroup

18.27.3.11 template<typename WeightUpdateModel , typename PostsynapticModel > SynapseGroup* NNmodel::addSynapsePopulation (const string & *name*, SynapseMatrixType *mtype*, unsigned int *delaySteps*, const string & *src*, const string & *trg*, const typename WeightUpdateModel::ParamValues & *weightParamValues*, const typename WeightUpdateModel::VarValues & *weightVarInitialisers*, const typename PostsynapticModel::ParamValues & *postsynapticParamValues*, const typename PostsynapticModel::VarValues & *postsynapticVarInitialisers*) [inline]

Adds a synapse population to the model using singleton weight update and postsynaptic models created using standard DECLARE_MODEL and IMPLEMENT_MODEL macros.

Template Parameters

WeightUpdateModel	type of weight update model (derived from WeightUpdateModels::Base).
PostsynapticModel	type of postsynaptic model (derived from PostsynapticModels::Base).

Parameters

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.

Returns

pointer to newly created SynapseGroup

18.27.3.12 bool NNmodel::canRunOnCPU ( ) const

Can this model run on the CPU?

If we are running in CPU_ONLY mode this is always true, but some GPU functionality will prevent models being run on both CPU and GPU.

18.27.3.13 void NNmodel::finalize ( )

Declare that the model specification is finalised in modelDefinition().

18.27.3.14 const NeuronGroup ∗ NNmodel::findNeuronGroup (const std::string & *name*) const

Find a neuron group by name.

18.27.3.15 NeuronGroup ∗ NNmodel::findNeuronGroup (const std::string & *name*)

Find a neuron group by name.

18.27.3.16 const SynapseGroup ∗ NNmodel::findSynapseGroup (const std::string & *name*) const

Find a synapse group by name.

18.27.3.17 SynapseGroup ∗ NNmodel::findSynapseGroup (const std::string & *name*)

Find a synapse group by name.

18.27.3.18 double NNmodel::getDT ( ) const [inline]

Gets the model integration step size.

18.27.3.19 const std::string& NNmodel::getName ( ) const [inline]

Gets the name of the neuronal network model.

18.27.3.20 unsigned int NNmodel::getNeuronGridSize ( ) const

Gets the size of the neuron kernel thread grid.

This is calculated by adding together the number of threads required by each neuron population, padded to be a multiple of GPU's thread block size.

18.27.3.21 const map<string, NeuronGroup>& NNmodel::getNeuronGroups ( ) const [inline]

Get std::map containing all named NeuronGroup objects in model.

18.27.3.22 const map<string, string>& NNmodel::getNeuronKernelParameters ( ) const [inline]

Gets std::map containing names and types of each parameter that should be passed through to the neuron kernel.

18.27.3.23 unsigned int NNmodel::getNumNeurons ( ) const

How many neurons make up the entire model.

18.27.3.24 const std::string& NNmodel::getPrecision ( ) const [inline]

Gets the floating point numerical precision.

18.27.3.25 unsigned int NNmodel::getResetKernel ( ) const [inline]

Which kernel should contain the reset logic? Specified in terms of GENN_FLAGS.

18.27.3.26 const std::string& NNmodel::getRNType ( ) const [inline]

Gets the underlying type for random number generation (default: uint64_t)

18.27.3.27 unsigned int NNmodel::getSeed ( ) const [inline]

Get the random seed.

18.27.3.28 const map<string, string>& NNmodel::getSimLearnPostKernelParameters ( ) const [inline]

Gets std::map containing names and types of each parameter that should be passed through to the postsynaptic learning kernel.

18.27.3.29 unsigned int NNmodel::getSynapseDynamicsGridSize ( ) const

Gets the size of the synapse dynamics kernel thread grid.

This is calculated by adding together the number of threads required by each synapse population's synapse dynamics kernel, padded to be a multiple of GPU's thread block size.

18.27.3.30 const map<string, std::pair<unsigned int, unsigned int> >& NNmodel::getSynapseDynamicsGroups ( ) const [inline]

Get std::map containing names of synapse groups which require synapse dynamics and their thread IDs within the synapse dynamics kernel (padded to multiples of the GPU thread block size)

18.27.3.31 const map<string, string>& NNmodel::getSynapseDynamicsKernelParameters ( ) const [inline]

Gets std::map containing names and types of each parameter that should be passed through to the synapse dynamics kernel.

18.27.3.32 const map<string, SynapseGroup>& NNmodel::getSynapseGroups ( ) const [inline]

Get std::map containing all named SynapseGroup objects in model.

18.27.3.33 unsigned int NNmodel::getSynapseKernelGridSize ( ) const

Gets the size of the synapse kernel thread grid.

This is calculated by adding together the number of threads required by each synapse population's synapse kernel, padded to be a multiple of GPU's thread block size.

18.27.3.34 const map<string, string>& NNmodel::getSynapseKernelParameters ( ) const [inline]

Gets std::map containing names and types of each parameter that should be passed through to the synapse kernel.

18.27.3.35 unsigned int NNmodel::getSynapsePostLearnGridSize ( ) const

Gets the size of the post-synaptic learning kernel thread grid.

This is calculated by adding together the number of threads required by each synapse population's postsynaptic learning kernel, padded to be a multiple of GPU's thread block size.

18.27.3.36 const map<string, std::pair<unsigned int, unsigned int> >& NNmodel::getSynapsePostLearnGroups ( ) const [inline]

Get std::map containing names of synapse groups which require postsynaptic learning and their thread IDs within the postsynaptic learning kernel (padded to multiples of the GPU thread block size)

18.27.3.37 bool NNmodel::isDeviceInitRequired ( ) const

Does this model require device initialisation kernel.

NOTE this is for neuron groups and densely connected synapse groups only

18.27.3.38 bool NNmodel::isDeviceRNGRequired ( ) const

Do any populations or initialisation code in this model require a device RNG?

NOTE some model code will use per-neuron RNGs instead

18.27.3.39 bool NNmodel::isDeviceSparseInitRequired ( ) const

Does this model require a device sparse initialisation kernel.

NOTE this is for sparsely connected synapse groups only

18.27.3.40 bool NNmodel::isFinalized ( ) const [inline]

Is the model specification finalized.

18.27.3.41 bool NNmodel::isHostRNGRequired ( ) const

Do any populations or initialisation code in this model require a host RNG?

18.27.3.42 bool NNmodel::isSynapseGroupDynamicsRequired (const std::string & *name*) const

Does named synapse group have synapse dynamics.

18.27.3.43 bool NNmodel::isSynapseGroupPostLearningRequired (const std::string & *name*) const

Does named synapse group have post-synaptic learning.

18.27.3.44 bool NNmodel::isTimingEnabled ( ) const [inline]

Are timers and timing commands enabled.

18.27.3.45 string NNmodel::scalarExpr (const double *val*) const

Get the string literal that should be used to represent a value in the model's floating-point type.

18.27.3.46 void NNmodel::setConstInp (const string & , double)

This function has been deprecated in GeNN 2.2.

This function sets a global input value to the specified neuron group.

18.27.3.47 void NNmodel::setDT (double *newDT*)

Set the integration step size of the model.

This function sets the integration time step DT of the model.

18.27.3.48 void NNmodel::setGPUDevice (int *device*)

Sets the underlying type for random number generation (default: uint64_t)

This function defines the way how the GPU is chosen. If "AUTODEVICE" (-1) is given as the argument, GeNN will use internal heuristics to choose the device. Otherwise the argument is the device number and the indicated device will be used.

Method to choose the GPU to be used for the model. If "AUTODEVICE' (-1), GeNN will choose the device based on a heuristic rule.

18.27.3.49 void NNmodel::setMaxConn (const string & *sname,* unsigned int *maxConnP*)

This function defines the maximum number of connections for a neuron in the population.

18.27.3.50 void NNmodel::setName (const std::string &)

Method to set the neuronal network model name.

18.27.3.51 void NNmodel::setNeuronClusterIndex (const string & *neuronGroup,* int *hostID,* int *deviceID*)

Function for setting which host and which device a neuron group will be simulated on.

This function is for setting which host and which device a neuron group will be simulated on.

Parameters

neuronGroup	Name of the neuron population
hostID	ID of the host
deviceID	ID of the device

18.27.3.52 void NNmodel::setPopulationSums ( )

Set the accumulated sums of lowest multiple of kernel block size $>=$ group sizes for all simulated groups.

Accumulate the sums and block-size-padded sums of all simulation groups.

This method saves the neuron numbers of the populations rounded to the next multiple of the block size as well as the sums s(i) = sum_{1...i} n_i of the rounded population sizes. These are later used to determine the branching structure for the generated neuron kernel code.

18.27.3.53 void NNmodel::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.

18.27.3.54 void NNmodel::setRNType (const std::string & *type*)

Sets the underlying type for random number generation (default: uint64_t)

18.27.3.55 void NNmodel::setSeed (unsigned int *inseed*)

Set the random seed (disables automatic seeding if argument not 0).

This function sets the random seed. If the passed argument is $>$ 0, automatic seeding is disabled. If the argument is 0, the underlying seed is obtained from the time() function.

Parameters

inseed	the new seed

18.27.3.56 void NNmodel::setSpanTypeToPre (const string & *sname*)

Method for switching the execution order of synapses to pre-to-post.

This function defines the execution order of the synapses in the kernels (0 : execute for every postsynaptic neuron 1: execute for every presynaptic neuron)

Parameters

sname	name of the synapse group to which to apply the pre-synaptic span type

18.27.3.57 void NNmodel::setSynapseClusterIndex (const string & *synapseGroup,* int *hostID,* int *deviceID*)

Function for setting which host and which device a synapse group will be simulated on.

This function is for setting which host and which device a synapse group will be simulated on.

Parameters

synapseGroup	Name of the synapse population
hostID	ID of the host
deviceID	ID of the device

18.27.3.58 void NNmodel::setSynapseG (const string & , double)

This function has been depreciated as of GeNN 2.2.

This functions sets the global value of the maximal synaptic conductance for a synapse population that was idfentified as conductance specifcation method "GLOBALG".

18.27.3.59 void NNmodel::setTiming (bool *theTiming*)

Set whether timers and timing commands are to be included.

This function sets a flag to determine whether timers and timing commands are to be included in generated code.

18.27.3.60 bool NNmodel::zeroCopyInUse ( ) const

Are any variables in any populations in this model using zero-copy memory?

The documentation for this class was generated from the following files:

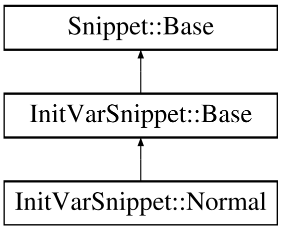
- modelSpec.h
- src/modelSpec.cc

18.28 InitVarSnippet::Normal Class Reference

Initialises variable by sampling from the normal distribution.

`#include <initVarSnippet.h>`

Inheritance diagram for InitVarSnippet::Normal:



Public Member Functions

- DECLARE_SNIPPET (InitVarSnippet::Normal, 2)
- SET_CODE ("$(value) = $(mean) + ($(gennrand_normal) ∗ $(sd));")
- virtual StringVec getParamNames () const

    Gets names of of (independent) model parameters.

Additional Inherited Members

18.28.1 Detailed Description

Initialises variable by sampling from the normal distribution.

This snippet takes 2 parameters:

- `mean` - The mean
- `sd` - The standard distribution

18.28.2 Member Function Documentation

18.28.2.1 InitVarSnippet::Normal::DECLARE_SNIPPET (InitVarSnippet::Normal , 2)

18.28.2.2 virtual StringVec InitVarSnippet::Normal::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

18.28.2.3 InitVarSnippet::Normal::SET_CODE ( )

The documentation for this class was generated from the following file:

- initVarSnippet.h

18.29 CodeStream::OB Struct Reference

An open bracket marker.

`#include <codeStream.h>`

Public Member Functions

- OB (unsigned int level)

Public Attributes

- const unsigned int Level

18.29.1 Detailed Description

An open bracket marker.

Write to code stream `os` using:

```
os << OB(16);
```

18.29.2 Constructor & Destructor Documentation

18.29.2.1 CodeStream::OB::OB (unsigned int *level*) [inline]

18.29.3 Member Data Documentation

18.29.3.1 const unsigned int CodeStream::OB::Level

The documentation for this struct was generated from the following file:

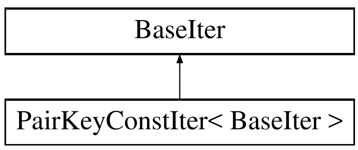
- codeStream.h

18.30 PairKeyConstIter< BaseIter > Class Template Reference

Custom iterator for iterating through the keys of containers containing pairs.

```
#include <codeGenUtils.h>
```

Inheritance diagram for PairKeyConstIter< BaseIter >:



Public Member Functions

- PairKeyConstIter ()
- PairKeyConstIter (BaseIter iter)
- const KeyType * operator-> () const
- const KeyType & operator* () const

18.30.1 Detailed Description

template<typename BaseIter>
class PairKeyConstIter< BaseIter >

Custom iterator for iterating through the keys of containers containing pairs.

18.30.2 Constructor & Destructor Documentation

18.30.2.1 template<typename BaseIter > PairKeyConstIter< BaseIter >::PairKeyConstIter ( ) [inline]

18.30.2.2 template<typename BaseIter > PairKeyConstIter< BaseIter >::PairKeyConstIter (BaseIter *iter*) [inline]

18.30.3 Member Function Documentation

18.30.3.1 template<typename BaseIter > const KeyType& PairKeyConstIter< BaseIter >::operator* ( ) const [inline]

18.30.3.2 template<typename BaseIter > const KeyType* PairKeyConstIter< BaseIter >::operator-> ( ) const [inline]

The documentation for this class was generated from the following file:

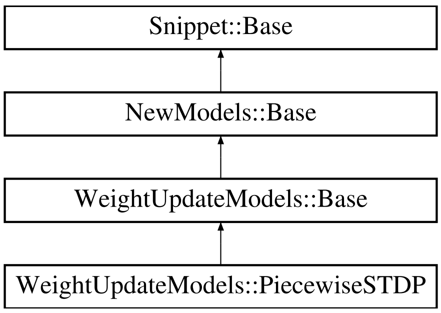
- codeGenUtils.h

18.31 WeightUpdateModels::PiecewiseSTDP Class Reference

This is a simple STDP rule including a time delay for the finite transmission speed of the synapse.

`#include <newWeightUpdateModels.h>`

Inheritance diagram for WeightUpdateModels::PiecewiseSTDP:



Public Types

- typedef Snippet::ValueBase< 10 > ParamValues
- typedef NewModels::VarInitContainerBase< 2 > VarValues

Public Member Functions

- virtual StringVec getParamNames () const

  Gets names of of (independent) model parameters.
- virtual StringPairVec 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.
- virtual DerivedParamVec getDerivedParams () const
- virtual bool isPreSpikeTimeRequired () const

Whether presynaptic spike times are needed or not.

- virtual bool isPostSpikeTimeRequired () const

  Whether postsynaptic spike times are needed or not.

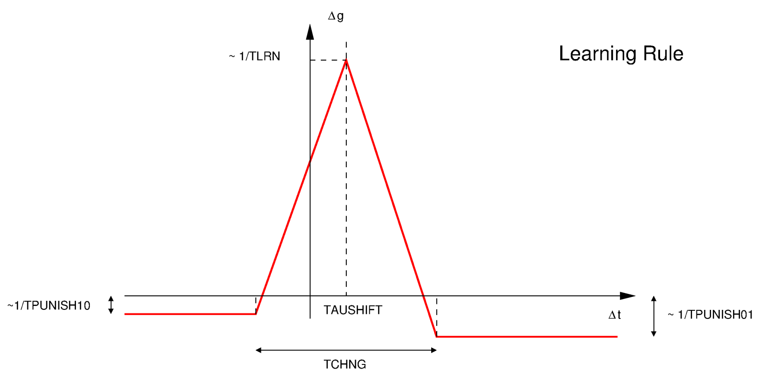
Static Public Member Functions

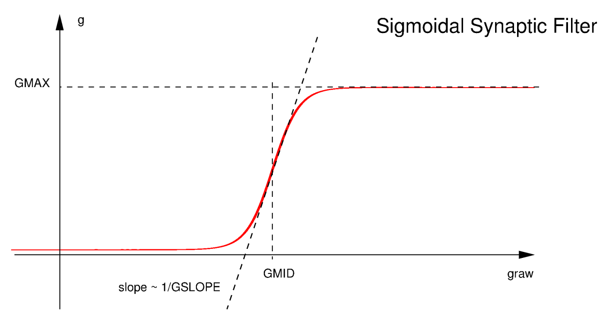
- static const PiecewiseSTDP ∗ getInstance ()

18.31.1 Detailed Description

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`.

Note

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
       \n", 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;
```

Note

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

18.31.2 Member Typedef Documentation

18.31.2.1 typedef Snippet::ValueBase< 10 > WeightUpdateModels::PiecewiseSTDP::ParamValues

18.31.2.2 typedef NewModels::VarInitContainerBase< 2 > WeightUpdateModels::PiecewiseSTDP::VarValues

18.31.3 Member Function Documentation

18.31.3.1 virtual DerivedParamVec WeightUpdateModels::PiecewiseSTDP::getDerivedParams ( ) const [inline], [virtual]

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

Reimplemented from Snippet::Base.

18.31.3.2 static const PiecewiseSTDP* WeightUpdateModels::PiecewiseSTDP::getInstance ( ) [inline], [static]

18.31.3.3 virtual std::string WeightUpdateModels::PiecewiseSTDP::getLearnPostCode ( ) const [inline], [virtual]

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.

Reimplemented from WeightUpdateModels::Base.

18.31.3.4 virtual StringVec WeightUpdateModels::PiecewiseSTDP::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

18.31.3.5 virtual std::string WeightUpdateModels::PiecewiseSTDP::getSimCode ( ) const [inline], [virtual]

Gets simulation code run when 'true' spikes are received.

Reimplemented from WeightUpdateModels::Base.

18.31.3.6 virtual StringPairVec WeightUpdateModels::PiecewiseSTDP::getVars ( ) const [inline], [virtual]

Gets names and types (as strings) of model variables.

Reimplemented from NewModels::Base.

18.31.3.7 virtual bool WeightUpdateModels::PiecewiseSTDP::isPostSpikeTimeRequired ( ) const [inline], [virtual]

Whether postsynaptic spike times are needed or not.

Reimplemented from WeightUpdateModels::Base.

18.31.3.8 virtual bool WeightUpdateModels::PiecewiseSTDP::isPreSpikeTimeRequired ( ) const [inline], [virtual]

Whether presynaptic spike times are needed or not.

Reimplemented from WeightUpdateModels::Base.

The documentation for this class was generated from the following file:

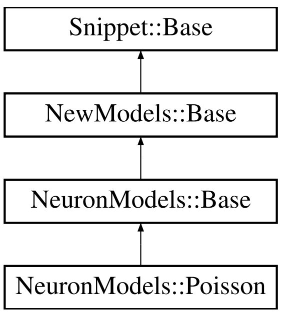
- newWeightUpdateModels.h

18.32 NeuronModels::Poisson Class Reference

Poisson neurons.

`#include <newNeuronModels.h>`

Inheritance diagram for NeuronModels::Poisson:



Public Types

- typedef Snippet::ValueBase< 4 > ParamValues
- typedef NewModels::VarInitContainerBase< 3 > VarValues

Public Member Functions

- virtual std::string getSimCode () const
  Gets the code that defines the execution of one timestep of integration of the neuron model.
- virtual std::string getThresholdConditionCode () const
  Gets code which defines the condition for a true spike in the described neuron model.
- virtual StringVec getParamNames () const
  Gets names of of (independent) model parameters.
- virtual StringPairVec getVars () const
  Gets names and types (as strings) of model variables.
- virtual StringPairVec getExtraGlobalParams () const
- virtual bool isPoisson () const

Static Public Member Functions

- static const NeuronModels::Poisson ∗ getInstance ()

18.32.1 Detailed Description

Poisson neurons.

Poisson neurons have constant membrane potential (`Vrest`) unless they are activated randomly to the `Vspike` value if (t- `SpikeTime`) > `trefract`.

It has 3 variables:

- `V` - Membrane potential
- `Seed` - Seed for random number generation
- `SpikeTime` - Time at which the neuron spiked for the last time

and 4 parameters:

- `therate` - Firing rate
- `trefract` - Refractory period
- `Vspike` - Membrane potential at spike (mV)
- `Vrest` - Membrane potential at rest (mV)

Note

The initial values array for the `Poisson` type needs three entries for `V`, `Seed` and `SpikeTime` and the parameter array needs four entries for `therate`, `trefract`, `Vspike` and `Vrest`, *in that order.*
Internally, GeNN 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 `therate` 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. An unrelated problem may occur with very low firing rates and small time steps. In that case it can occur that the firing probability is so small that the granularity of the 64 bit integer based random number generator begins to show. The effect manifests itself in that small changes in the firing rate do not seem to have an effect on the behaviour of the Poisson neurons because the numbers are so small that only if the random number is identical 0 a spike will be triggered.
GeNN uses a separate random number generator for each Poisson neuron. The seeds (and later states) of these random number generators are stored in the `seed` variable. GeNN allocates memory for these seeds/states in the generated `allocateMem()` function. It is, however, currently the responsibility of the user to fill the array of seeds with actual random seeds. Not doing so carries the risk that all random number generators are seeded with the same seed ("0") and produce the same random numbers across neurons at each given time step. When using the GPU, `seed` also must be copied to the GPU after having been initialized.

18.32.2 Member Typedef Documentation

18.32.2.1 typedef Snippet::ValueBase< 4 > NeuronModels::Poisson::ParamValues

18.32.2.2 typedef NewModels::VarInitContainerBase< 3 > NeuronModels::Poisson::VarValues

18.32.3 Member Function Documentation

18.32.3.1 virtual StringPairVec NeuronModels::Poisson::getExtraGlobalParams ( ) const [inline], [virtual]

Gets names and types (as strings) of additional per-population parameters for the weight update model.

Reimplemented from NeuronModels::Base.

18.32.3.2 static const NeuronModels::Poisson* NeuronModels::Poisson::getInstance ( ) [inline], [static]

18.32.3.3 virtual StringVec NeuronModels::Poisson::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

18.32.3.4 virtual std::string NeuronModels::Poisson::getSimCode ( ) const [inline], [virtual]

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.

Reimplemented from NeuronModels::Base.

18.32.3.5 virtual std::string NeuronModels::Poisson::getThresholdConditionCode ( ) const [inline], [virtual]

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

Reimplemented from NeuronModels::Base.

18.32.3.6 virtual StringPairVec NeuronModels::Poisson::getVars ( ) const [inline], [virtual]

Gets names and types (as strings) of model variables.

Reimplemented from NewModels::Base.

18.32.3.7 virtual bool NeuronModels::Poisson::isPoisson ( ) const [inline], [virtual]

Is this neuron model the internal Poisson model (which requires a number of special cases)

Reimplemented from NeuronModels::Base.

The documentation for this class was generated from the following file:

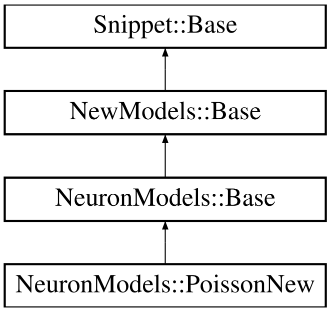
- newNeuronModels.h

18.33 NeuronModels::PoissonNew Class Reference

Poisson neurons.

`#include <newNeuronModels.h>`

Inheritance diagram for NeuronModels::PoissonNew:



Public Types

- typedef Snippet::ValueBase$<$ 1 $>$ ParamValues
- typedef NewModels::VarInitContainerBase$<$ 1 $>$ VarValues

Public Member Functions

- virtual std::string getSimCode () const

  Gets the code that defines the execution of one timestep of integration of the neuron model.
- virtual std::string getThresholdConditionCode () const

  Gets code which defines the condition for a true spike in the described neuron model.
- virtual StringVec getParamNames () const

  Gets names of of (independent) model parameters.
- virtual StringPairVec getVars () const

  Gets names and types (as strings) of model variables.
- virtual DerivedParamVec getDerivedParams () const

Static Public Member Functions

- static const NeuronModels::PoissonNew $*$ getInstance ()

18.33.1 Detailed Description

Poisson neurons.

It has 1 state variable:

- `timeStepToSpike` - Number of timesteps to next spike

and 1 parameter:

- `rate` - Mean firing rate (Hz)

Note

Internally this samples from the exponential distribution using the C++ 11 <random> library on the CPU and Von Neumann's exponential generator (Ripley p.230) implemented using cuRAND on the GPU.

18.33.2 Member Typedef Documentation

18.33.2.1 typedef Snippet::ValueBase< 1 > NeuronModels::PoissonNew::ParamValues

18.33.2.2 typedef NewModels::VarInitContainerBase< 1 > NeuronModels::PoissonNew::VarValues

18.33.3 Member Function Documentation

18.33.3.1 virtual DerivedParamVec NeuronModels::PoissonNew::getDerivedParams ( ) const [inline], [virtual]

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

Reimplemented from Snippet::Base.

18.33.3.2 static const NeuronModels::PoissonNew* NeuronModels::PoissonNew::getInstance ( ) [inline], [static]

18.33.3.3 virtual StringVec NeuronModels::PoissonNew::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

18.33.3.4 virtual std::string NeuronModels::PoissonNew::getSimCode ( ) const [inline], [virtual]

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.

Reimplemented from NeuronModels::Base.

18.33.3.5 virtual std::string NeuronModels::PoissonNew::getThresholdConditionCode ( ) const [inline], [virtual]

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

Reimplemented from NeuronModels::Base.

18.33.3.6 virtual StringPairVec NeuronModels::PoissonNew::getVars ( ) const [inline], [virtual]

Gets names and types (as strings) of model variables.

Reimplemented from NewModels::Base.

The documentation for this class was generated from the following file:

- newNeuronModels.h

18.34 postSynModel Class Reference

Class to hold the information that defines a post-synaptic model (a model of how synapses affect post-synaptic neuron variables, classically in the form of a synaptic current). It also allows to define an equation for the dynamics that can be applied to the summed synaptic input variable "insyn".

`#include <postSynapseModels.h>`

Public Member Functions

- postSynModel ()

  Constructor for postSynModel objects.
- ∼postSynModel ()

  Destructor for postSynModel objects.

Public Attributes

- string postSyntoCurrent

  Code that defines how postsynaptic update is translated to current.
- string postSynDecay

  Code that defines how postsynaptic current decays.
- string supportCode

  Support code is made available within the neuron kernel definition file and is meant 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.
- vector< string > varNames

  Names of the variables in the postsynaptic model.
- vector< string > varTypes

  Types of the variable named above, e.g. "float". Names and types are matched by their order of occurrence in the vector.
- vector< string > pNames

  Names of (independent) parameters of the model.
- vector< string > dpNames

  Names of dependent parameters of the model.
- dpclass ∗ dps

  Derived parameters.

18.34.1 Detailed Description

Class to hold the information that defines a post-synaptic model (a model of how synapses affect post-synaptic neuron variables, classically in the form of a synaptic current). It also allows to define an equation for the dynamics that can be applied to the summed synaptic input variable "insyn".

18.34.2 Constructor & Destructor Documentation

18.34.2.1 postSynModel::postSynModel ( )

Constructor for postSynModel objects.

18.34.2.2 postSynModel::∼postSynModel ( )

Destructor for postSynModel objects.

18.34.3 Member Data Documentation

18.34.3.1 vector<string> postSynModel::dpNames

Names of dependent parameters of the model.

18.34.3.2 dpclass* postSynModel::dps

Derived parameters.

18.34.3.3 vector< string > postSynModel::pNames

Names of (independent) parameters of the model.

18.34.3.4 string postSynModel::postSynDecay

Code that defines how postsynaptic current decays.

18.34.3.5 string postSynModel::postSyntoCurrent

Code that defines how postsynaptic update is translated to current.

18.34.3.6 string postSynModel::supportCode

Support code is made available within the neuron kernel definition file and is meant 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.

18.34.3.7 vector< string > postSynModel::varNames

Names of the variables in the postsynaptic model.

18.34.3.8 vector< string > postSynModel::varTypes

Types of the variable named above, e.g. "float". Names and types are matched by their order of occurrence in the vector.

The documentation for this class was generated from the following files:

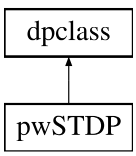
- postSynapseModels.h
- postSynapseModels.cc

18.35 pwSTDP Class Reference

TODO This class definition may be code-generated in a future release.

```
#include <synapseModels.h>
```

Inheritance diagram for pwSTDP:



Public Member Functions

- double calculateDerivedParameter (int index, vector< double > pars, double=1.0)

18.35.1 Detailed Description

TODO This class definition may be code-generated in a future release.

This class defines derived parameters for the learn1synapse standard weightupdate model

18.35.2 Member Function Documentation

18.35.2.1 double pwSTDP::calculateDerivedParameter (int *index*, vector< double > *pars*, double = 1.0) [inline], [virtual]

Reimplemented from dpclass.

The documentation for this class was generated from the following file:

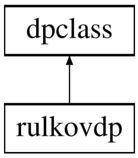
- synapseModels.h

18.36 rulkovdp Class Reference

Class defining the dependent parameters of the Rulkov map neuron.

```
#include <neuronModels.h>
```

Inheritance diagram for rulkovdp:



Public Member Functions

- double calculateDerivedParameter (int index, vector< double > pars, double=1.0)

18.36.1 Detailed Description

Class defining the dependent parameters of the Rulkov map neuron.

18.36.2 Member Function Documentation

18.36.2.1 double rulkovdp::calculateDerivedParameter (int *index*, vector< double > *pars*, double = 1.0) [inline], [virtual]

Reimplemented from dpclass.

The documentation for this class was generated from the following file:

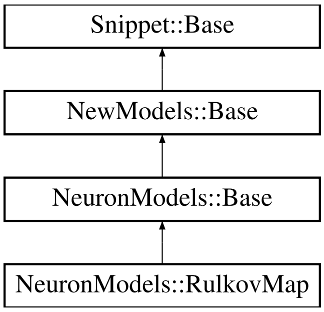
- neuronModels.h

18.37 NeuronModels::RulkovMap Class Reference

Rulkov Map neuron.

```
#include <newNeuronModels.h>
```

Inheritance diagram for NeuronModels::RulkovMap:



Public Types

- typedef Snippet::ValueBase< 4 > ParamValues
- typedef NewModels::VarInitContainerBase< 2 > VarValues

Public Member Functions

- virtual std::string getSimCode () const

  Gets the code that defines the execution of one timestep of integration of the neuron model.
- virtual std::string getThresholdConditionCode () const

  Gets code which defines the condition for a true spike in the described neuron model.
- virtual StringVec getParamNames () const

  Gets names of of (independent) model parameters.
- virtual StringPairVec getVars () const

  Gets names and types (as strings) of model variables.
- virtual DerivedParamVec getDerivedParams () const

Static Public Member Functions

- static const NeuronModels::RulkovMap ∗ getInstance ()

18.37.1 Detailed Description

Rulkov Map neuron.

The RulkovMap type is a map based neuron model based on [4] but in the 1-dimensional map form used in [3] :

$$V(t+\Delta t) = \begin{cases} V_{\rm spike}\left(\frac{\alpha V_{\rm spike}}{V_{\rm spike}-V(t)\beta I_{\rm syn}}+y\right) & V(t) \leq 0 \\ V_{\rm spike}\left(\alpha+y\right) & V(t) \leq V_{\rm spike}\left(\alpha+y\right) \;\&\; V(t-\Delta t) \leq 0 \\ -V_{\rm spike} & \text{otherwise} \end{cases}$$

Note

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 $\mathrm{M}\Omega$.

Note

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*.

18.37.2 Member Typedef Documentation

18.37.2.1 typedef Snippet::ValueBase< 4 > NeuronModels::RulkovMap::ParamValues

18.37.2.2 typedef NewModels::VarInitContainerBase< 2 > NeuronModels::RulkovMap::VarValues

18.37.3 Member Function Documentation

18.37.3.1 virtual DerivedParamVec NeuronModels::RulkovMap::getDerivedParams ( ) const [inline], [virtual]

Gets names of derived model parameters and the function objects to call to Calculate their value from a vector of model parameter values

Reimplemented from Snippet::Base.

18.37.3.2 static const NeuronModels::RulkovMap∗ NeuronModels::RulkovMap::getInstance ( ) [inline], [static]

18.37.3.3 virtual StringVec NeuronModels::RulkovMap::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

18.37.3.4 virtual std::string NeuronModels::RulkovMap::getSimCode ( ) const [inline], [virtual]

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.

Reimplemented from NeuronModels::Base.

18.37.3.5 virtual std::string NeuronModels::RulkovMap::getThresholdConditionCode ( ) const [inline], [virtual]

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

Reimplemented from NeuronModels::Base.

18.37.3.6 virtual StringPairVec NeuronModels::RulkovMap::getVars ( ) const [inline], [virtual]

Gets names and types (as strings) of model variables.

Reimplemented from NewModels::Base.

The documentation for this class was generated from the following file:

- newNeuronModels.h

18.38 SparseProjection Struct Reference

class (struct) for defining a spars connectivity projection

`#include <sparseProjection.h>`

Public Attributes

- unsigned int ∗ indInG
- unsigned int ∗ ind
- unsigned int ∗ preInd
- unsigned int ∗ revIndInG
- unsigned int ∗ revInd
- unsigned int ∗ remap
- unsigned int connN

18.38.1 Detailed Description

class (struct) for defining a spars connectivity projection

18.38.2 Member Data Documentation

18.38.2.1 unsigned int SparseProjection::connN

18.38.2.2 unsigned int∗ SparseProjection::ind

18.38.2.3 unsigned int∗ SparseProjection::indInG

18.38.2.4 unsigned int∗ SparseProjection::preInd

18.38.2.5 unsigned int∗ SparseProjection::remap

18.38.2.6 unsigned int∗ SparseProjection::revInd

18.38.2.7 unsigned int∗ SparseProjection::revIndInG

The documentation for this struct was generated from the following file:

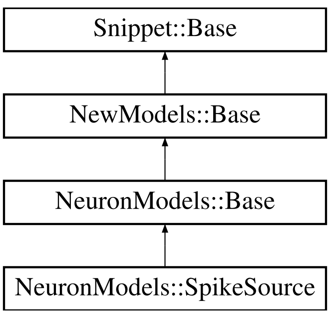
- sparseProjection.h

18.39 NeuronModels::SpikeSource Class Reference

Empty neuron which allows setting spikes from external sources.

`#include <newNeuronModels.h>`

Inheritance diagram for NeuronModels::SpikeSource:



Public Types

- typedef Snippet::ValueBase< 0 > ParamValues
- typedef NewModels::VarInitContainerBase< 0 > VarValues

Public Member Functions

- virtual std::string getThresholdConditionCode () const

    Gets code which defines the condition for a true spike in the described neuron model.

Static Public Member Functions

- static const NeuronModels::SpikeSource ∗ getInstance ()

18.39.1 Detailed Description

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 SpikeGenerator↩ Group in Brian or a SpikeSourceArray in PyNN.

18.39.2 Member Typedef Documentation

18.39.2.1 typedef **Snippet::ValueBase**< 0 > **NeuronModels::SpikeSource::ParamValues**

18.39.2.2 typedef **NewModels::VarInitContainerBase**< 0 > **NeuronModels::SpikeSource::VarValues**

18.39.3 Member Function Documentation

18.39.3.1 static const **NeuronModels::SpikeSource**∗ NeuronModels::SpikeSource::getInstance ( ) `[inline]`, `[static]`

18.39.3.2 virtual std::string NeuronModels::SpikeSource::getThresholdConditionCode ( ) const `[inline]`, `[virtual]`

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

Reimplemented from NeuronModels::Base.

The documentation for this class was generated from the following file:

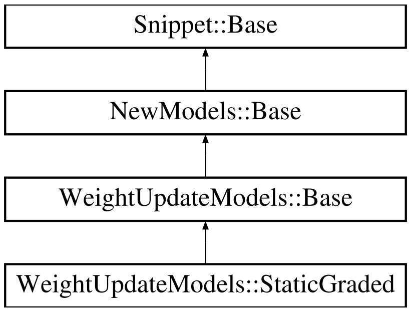
- newNeuronModels.h

18.40 WeightUpdateModels::StaticGraded Class Reference

Graded-potential, static synapse.

`#include <newWeightUpdateModels.h>`

Inheritance diagram for WeightUpdateModels::StaticGraded:



Public Types

- typedef Snippet::ValueBase< 2 > ParamValues
- typedef NewModels::VarInitContainerBase< 1 > VarValues

Public Member Functions

- virtual StringVec getParamNames () const

  Gets names of of (independent) model parameters.
- virtual StringPairVec 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.

Static Public Member Functions

- static const StaticGraded ∗ getInstance ()

18.40.1 Detailed Description

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));
$(updatelinsyn);
```

`event` threshold condition code is:

```
$(V_pre) > $(Epre)
```

Note

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`.

18.40.2 Member Typedef Documentation

18.40.2.1 typedef Snippet::ValueBase< 2 > WeightUpdateModels::StaticGraded::ParamValues

18.40.2.2 typedef NewModels::VarInitContainerBase< 1 > WeightUpdateModels::StaticGraded::VarValues

18.40.3 Member Function Documentation

18.40.3.1 virtual std::string WeightUpdateModels::StaticGraded::getEventCode ( ) const [inline], [virtual]

Gets code run when events (all the instances where event threshold condition is met) are received.

Reimplemented from WeightUpdateModels::Base.

18.40.3.2 virtual std::string WeightUpdateModels::StaticGraded::getEventThresholdConditionCode ( ) const [inline], [virtual]

Gets codes to test for events.

Reimplemented from WeightUpdateModels::Base.

18.40.3.3 static const StaticGraded* WeightUpdateModels::StaticGraded::getInstance ( ) [inline], [static]

18.40.3.4 virtual StringVec WeightUpdateModels::StaticGraded::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

18.40.3.5 virtual StringPairVec WeightUpdateModels::StaticGraded::getVars ( ) const [inline], [virtual]

Gets names and types (as strings) of model variables.

Reimplemented from NewModels::Base.

The documentation for this class was generated from the following file:

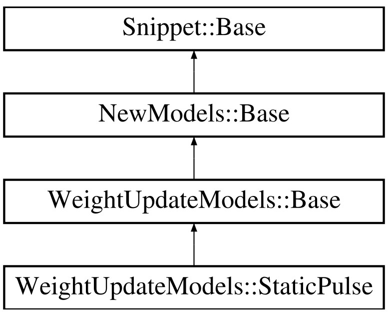
- newWeightUpdateModels.h

18.41 WeightUpdateModels::StaticPulse Class Reference

Pulse-coupled, static synapse.

`#include <newWeightUpdateModels.h>`

Inheritance diagram for WeightUpdateModels::StaticPulse:



Public Types

- typedef Snippet::ValueBase< 0 > ParamValues
- typedef NewModels::VarInitContainerBase< 1 > VarValues

Public Member Functions

- virtual StringPairVec 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.

Static Public Member Functions

- static const StaticPulse ∗ getInstance ()

18.41.1 Detailed Description

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);\n\
$(updatelinsyn);\n"
```

18.41.2 Member Typedef Documentation

18.41.2.1 typedef **Snippet::ValueBase**< 0 > **WeightUpdateModels::StaticPulse::ParamValues**

18.41.2.2 typedef **NewModels::VarInitContainerBase**< 1 > **WeightUpdateModels::StaticPulse::VarValues**

18.41.3 Member Function Documentation

18.41.3.1 static const **StaticPulse**∗ WeightUpdateModels::StaticPulse::getInstance ( ) `[inline]`, `[static]`

18.41.3.2 virtual std::string WeightUpdateModels::StaticPulse::getSimCode ( ) const `[inline]`, `[virtual]`

Gets simulation code run when 'true' spikes are received.

Reimplemented from WeightUpdateModels::Base.

18.41.3.3 virtual **StringPairVec** WeightUpdateModels::StaticPulse::getVars ( ) const `[inline]`, `[virtual]`

Gets names and types (as strings) of model variables.

Reimplemented from NewModels::Base.

The documentation for this class was generated from the following file:

- newWeightUpdateModels.h

18.42 stopWatch Struct Reference

`#include <hr_time.h>`

Public Attributes

- timeval start
- timeval stop

18.42.1 Member Data Documentation

18.42.1.1 timeval stopWatch::start

18.42.1.2 timeval stopWatch::stop

The documentation for this struct was generated from the following file:

- hr_time.h

18.43 SynapseGroup Class Reference

`#include <synapseGroup.h>`

Public Types

- enum SpanType { SpanType::POSTSYNAPTIC, SpanType::PRESYNAPTIC }

Public Member Functions

- SynapseGroup (const std::string name, SynapseMatrixType matrixType, unsigned int delaySteps, const WeightUpdateModels::Base ∗wu, const std::vector< double > &wuParams, const std::vector< NewModels::VarInit > &wuVarInitialisers, const PostsynapticModels::Base ∗ps, const std::vector< double > &psParams, const std::vector< NewModels::VarInit > &psVarInitialisers, NeuronGroup ∗srcNeuronGroup, NeuronGroup ∗trgNeuronGroup)
- SynapseGroup (const SynapseGroup &)=delete
- SynapseGroup ()=delete
- NeuronGroup ∗ getSrcNeuronGroup ()
- NeuronGroup ∗ getTrgNeuronGroup ()
- void setTrueSpikeRequired (bool req)
- void setSpikeEventRequired (bool req)
- void setEventThresholdReTestRequired (bool req)

  Function to enable the use of zero-copied memory for a particular weight update model state variable (deprecated use SynapseGroup::setWUVarMode):
- void setWUVarZeroCopyEnabled (const std::string &varName, bool enabled)

  Function to enable the use zero-copied memory for a particular postsynaptic model state variable (deprecated use SynapseGroup::setWUVarMode)
- void setPSVarZeroCopyEnabled (const std::string &varName, bool enabled)
- void setWUVarMode (const std::string &varName, VarMode mode)

  Set variable mode of weight update model state variable.
- void setPSVarMode (const std::string &varName, VarMode mode)

  Set variable mode of postsynaptic model state variable.
- void setClusterIndex (int hostID, int deviceID)
- void setInSynVarMode (VarMode mode)

Set variable mode used for variables used to combine input from this synapse group.

- void setMaxConnections (unsigned int maxConnections)

  Sets the maximum number of target neurons any source neurons can connect to.
- void setSpanType (SpanType spanType)

  Set how CUDA implementation is parallelised.
- void initDerivedParams (double dt)
- void calcKernelSizes (unsigned int blockSize, unsigned int &paddedKernelIDStart)
- std::pair< unsigned int, unsigned int > getPaddedKernelIDRange () const
- const std::string & getName () const
- SpanType getSpanType () const
- unsigned int getDelaySteps () const
- unsigned int getMaxConnections () const
- SynapseMatrixType getMatrixType () const
- VarMode getInSynVarMode () const

  Get variable mode used for variables used to combine input from this synapse group.
- unsigned int getPaddedDynKernelSize (unsigned int blockSize) const
- unsigned int getPaddedPostLearnKernelSize (unsigned int blockSize) const
- const NeuronGroup * getSrcNeuronGroup () const
- const NeuronGroup * getTrgNeuronGroup () const
- bool isTrueSpikeRequired () const
- bool isSpikeEventRequired () const
- bool isEventThresholdReTestRequired () const
- const WeightUpdateModels::Base * getWUModel () const
- const std::vector< double > & getWUParams () const
- const std::vector< double > & getWUDerivedParams () const
- const std::vector< NewModels::VarInit > & getWUVarInitialisers () const
- const std::vector< double > getWUConstInitVals () const
- const PostsynapticModels::Base * getPSModel () const
- const std::vector< double > & getPSParams () const
- const std::vector< double > & getPSDerivedParams () const
- const std::vector< NewModels::VarInit > & getPSVarInitialisers () const
- const std::vector< double > getPSConstInitVals () const
- bool isZeroCopyEnabled () const
- bool isWUVarZeroCopyEnabled (const std::string &var) const
- bool isPSVarZeroCopyEnabled (const std::string &var) const
- VarMode getWUVarMode (const std::string &var) const

  Get variable mode used by weight update model state variable.
- VarMode getWUVarMode (size_t index) const

  Get variable mode used by weight update model state variable.
- VarMode getPSVarMode (const std::string &var) const

  Get variable mode used by postsynaptic model state variable.
- VarMode getPSVarMode (size_t index) const

  Get variable mode used by postsynaptic model state variable.
- bool isPSAtomicAddRequired (unsigned int blockSize) const

  Is this synapse group too large to use shared memory for combining postsynaptic output.
- void addExtraGlobalNeuronParams (std::map< string, string > &kernelParameters) const
- void addExtraGlobalSynapseParams (std::map< string, string > &kernelParameters) const
- void addExtraGlobalPostLearnParams (std::map< string, string > &kernelParameters) const
- void addExtraGlobalSynapseDynamicsParams (std::map< string, string > &kernelParameters) const
- std::string getOffsetPre () const
- std::string getOffsetPost (const std::string &devPrefix) const
- bool isPSInitRNGRequired (VarInit varInitMode) const

  Does this synapse group require an RNG for it's postsynaptic init code.

- bool isWUInitRNGRequired (VarInit varInitMode) const

  Does this synapse group require an RNG for it's weight update init code.
- bool isPSDeviceVarInitRequired () const

  Is device var init code required for any variables in this synapse group's postsynaptic model.
- bool isWUDeviceVarInitRequired () const

  Is device var init code required for any variables in this synapse group's weight update model.
- bool canRunOnCPU () const

  Can this synapse group run on the CPU?

18.43.1 Member Enumeration Documentation

18.43.1.1 enum SynapseGroup::SpanType [strong]

Enumerator

POSTSYNAPTIC

PRESYNAPTIC

18.43.2 Constructor & Destructor Documentation

18.43.2.1 SynapseGroup::SynapseGroup (const std::string *name*, SynapseMatrixType *matrixType*, unsigned int *delaySteps*, const WeightUpdateModels::Base ∗ *wu*, const std::vector< double > & *wuParams*, const std::vector< NewModels::VarInit > & *wuVarInitialisers*, const PostsynapticModels::Base ∗ *ps*, const std::vector< double > & *psParams*, const std::vector< NewModels::VarInit > & *psVarInitialisers*, NeuronGroup ∗ *srcNeuronGroup*, NeuronGroup ∗ *trgNeuronGroup*)

18.43.2.2 SynapseGroup::SynapseGroup (const SynapseGroup &) [delete]

18.43.2.3 SynapseGroup::SynapseGroup ( ) [delete]

18.43.3 Member Function Documentation

18.43.3.1 void SynapseGroup::addExtraGlobalNeuronParams (std::map< string, string > & *kernelParameters*) const

18.43.3.2 void SynapseGroup::addExtraGlobalPostLearnParams (std::map< string, string > & *kernelParameters*) const

18.43.3.3 void SynapseGroup::addExtraGlobalSynapseDynamicsParams (std::map< string, string > & *kernelParameters*) const

18.43.3.4 void SynapseGroup::addExtraGlobalSynapseParams (std::map< string, string > & *kernelParameters*) const

18.43.3.5 void SynapseGroup::calcKernelSizes (unsigned int *blockSize*, unsigned int & *paddedKernelIDStart*)

18.43.3.6 bool SynapseGroup::canRunOnCPU ( ) const

Can this synapse group run on the CPU?

If we are running in CPU_ONLY mode this is always true, but some GPU functionality will prevent models being run on both CPU and GPU.

18.43.3.7 unsigned int SynapseGroup::getDelaySteps ( ) const [inline]

18.43.3.8 VarMode SynapseGroup::getInSynVarMode ( ) const [inline]

Get variable mode used for variables used to combine input from this synapse group.

18.43.3.9 SynapseMatrixType SynapseGroup::getMatrixType ( ) const [inline]

18.43.3.10 unsigned int SynapseGroup::getMaxConnections ( ) const [inline]

18.43.3.11 const std::string& SynapseGroup::getName ( ) const [inline]

18.43.3.12 std::string SynapseGroup::getOffsetPost (const std::string & *devPrefix*) const

18.43.3.13 std::string SynapseGroup::getOffsetPre ( ) const

18.43.3.14 unsigned int SynapseGroup::getPaddedDynKernelSize (unsigned int *blockSize*) const

18.43.3.15 std::pair<unsigned int, unsigned int> SynapseGroup::getPaddedKernelIDRange ( ) const [inline]

18.43.3.16 unsigned int SynapseGroup::getPaddedPostLearnKernelSize (unsigned int *blockSize*) const

18.43.3.17 const std::vector< double > SynapseGroup::getPSConstInitVals ( ) const

18.43.3.18 const std::vector<double>& SynapseGroup::getPSDerivedParams ( ) const [inline]

18.43.3.19 const PostsynapticModels::Base* SynapseGroup::getPSModel ( ) const [inline]

18.43.3.20 const std::vector<double>& SynapseGroup::getPSParams ( ) const [inline]

18.43.3.21 const std::vector<NewModels::VarInit>& SynapseGroup::getPSVarInitialisers ( ) const [inline]

18.43.3.22 VarMode SynapseGroup::getPSVarMode (const std::string & *var*) const

Get variable mode used by postsynaptic model state variable.

18.43.3.23 VarMode SynapseGroup::getPSVarMode (size_t *index*) const [inline]

Get variable mode used by postsynaptic model state variable.

18.43.3.24 SpanType SynapseGroup::getSpanType ( ) const [inline]

18.43.3.25 NeuronGroup* SynapseGroup::getSrcNeuronGroup ( ) [inline]

18.43.3.26 const NeuronGroup* SynapseGroup::getSrcNeuronGroup ( ) const [inline]

18.43.3.27 NeuronGroup* SynapseGroup::getTrgNeuronGroup ( ) [inline]

18.43.3.28 const NeuronGroup* SynapseGroup::getTrgNeuronGroup ( ) const [inline]

18.43.3.29 const std::vector< double > SynapseGroup::getWUConstInitVals ( ) const

18.43.3.30 const std::vector<double>& SynapseGroup::getWUDerivedParams ( ) const [inline]

18.43.3.31 const WeightUpdateModels::Base* SynapseGroup::getWUModel ( ) const [inline]

18.43.3.32 const std::vector<double>& SynapseGroup::getWUParams ( ) const [inline]

18.43.3.33 const std::vector<NewModels::VarInit>& SynapseGroup::getWUVarInitialisers ( ) const [inline]

18.43.3.34 VarMode SynapseGroup::getWUVarMode (const std::string & *var*) const

Get variable mode used by weight update model state variable.

18.43.3.35 VarMode SynapseGroup::getWUVarMode (size_t *index*) const [inline]

Get variable mode used by weight update model state variable.

18.43.3.36 void SynapseGroup::initDerivedParams (double *dt*)

18.43.3.37 bool SynapseGroup::isEventThresholdReTestRequired ( ) const [inline]

18.43.3.38 bool SynapseGroup::isPSAtomicAddRequired (unsigned int *blockSize*) const

Is this synapse group too large to use shared memory for combining postsynaptic output.

18.43.3.39 bool SynapseGroup::isPSDeviceVarInitRequired ( ) const

Is device var init code required for any variables in this synapse group's postsynaptic model.

18.43.3.40 bool SynapseGroup::isPSInitRNGRequired (VarInit *varInitMode*) const

Does this synapse group require an RNG for it's postsynaptic init code.

18.43.3.41 bool SynapseGroup::isPSVarZeroCopyEnabled (const std::string & *var*) const [inline]

18.43.3.42 bool SynapseGroup::isSpikeEventRequired ( ) const [inline]

18.43.3.43 bool SynapseGroup::isTrueSpikeRequired ( ) const [inline]

18.43.3.44 bool SynapseGroup::isWUDeviceVarInitRequired ( ) const

Is device var init code required for any variables in this synapse group's weight update model.

18.43.3.45 bool SynapseGroup::isWUInitRNGRequired (VarInit *varInitMode*) const

Does this synapse group require an RNG for it's weight update init code.

18.43.3.46 bool SynapseGroup::isWUVarZeroCopyEnabled (const std::string & *var*) const [inline]

18.43.3.47 bool SynapseGroup::isZeroCopyEnabled ( ) const

18.43.3.48 void SynapseGroup::setClusterIndex (int *hostID*, int *deviceID*) [inline]

18.43.3.49 void SynapseGroup::setEventThresholdReTestRequired (bool *req*) [inline]

Function to enable the use of zero-copied memory for a particular weight update model state variable (deprecated use SynapseGroup::setWUVarMode):

18.43.3.50 void SynapseGroup::setInSynVarMode (VarMode *mode*) [inline]

Set variable mode used for variables used to combine input from this synapse group.

This is ignored for CPU simulations

18.43.3.51 void SynapseGroup::setMaxConnections (unsigned int *maxConnections*)

Sets the maximum number of target neurons any source neurons can connect to.

Use with SynapseMatrixType::SPARSE_GLOBALG and SynapseMatrixType::SPARSE_INDIVIDUALG to optimise CUDA implementation

18.43.3.52 void SynapseGroup::setPSVarMode (const std::string & *varName*, VarMode *mode*)

Set variable mode of postsynaptic model state variable.

This is ignored for CPU simulations

18.43.3.53 void SynapseGroup::setPSVarZeroCopyEnabled (const std::string & *varName*, bool *enabled*) [inline]

May improve IO performance at the expense of kernel performance

18.43.3.54 void SynapseGroup::setSpanType (SpanType *spanType*)

Set how CUDA implementation is parallelised.

with a thread per target neuron (default) or a thread per source spike

18.43.3.55 void SynapseGroup::setSpikeEventRequired (bool *req*) [inline]

18.43.3.56 void SynapseGroup::setTrueSpikeRequired (bool *req*) [inline]

18.43.3.57 void SynapseGroup::setWUVarMode (const std::string & *varName*, VarMode *mode*)

Set variable mode of weight update model state variable.

This is ignored for CPU simulations

18.43.3.58 void SynapseGroup::setWUVarZeroCopyEnabled (const std::string & *varName*, bool *enabled*) [inline]

Function to enable the use zero-copied memory for a particular postsynaptic model state variable (deprecated use SynapseGroup::setWUVarMode)

May improve IO performance at the expense of kernel performance

The documentation for this class was generated from the following files:

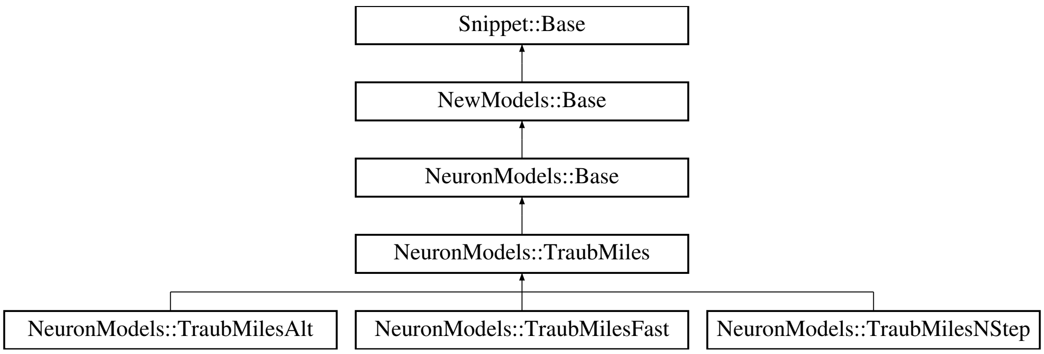
- synapseGroup.h
- synapseGroup.cc

18.44 NeuronModels::TraubMiles Class Reference

Hodgkin-Huxley neurons with Traub & Miles algorithm.

```
#include <newNeuronModels.h>
```

Inheritance diagram for NeuronModels::TraubMiles:



Public Types

- typedef Snippet::ValueBase< 7 > ParamValues
- typedef NewModels::VarInitContainerBase< 4 > VarValues

Public Member Functions

- virtual std::string getSimCode () const

  Gets the code that defines the execution of one timestep of integration of the neuron model.
- virtual std::string getThresholdConditionCode () const

  Gets code which defines the condition for a true spike in the described neuron model.

- virtual StringVec getParamNames () const

    Gets names of of (independent) model parameters.
- virtual StringPairVec getVars () const

    Gets names and types (as strings) of model variables.

Static Public Member Functions

- static const NeuronModels::TraubMiles ∗ getInstance ()

18.44.1 Detailed Description

Hodgkin-Huxley neurons with Traub & Miles algorithm.

This conductance based model has been taken from [5] and can be described by the equations:

$$\begin{aligned} C\frac{dV}{dt} &= -I_{\rm Na} - I_K - I_{\rm leak} - I_M - I_{i,DC} - I_{i,{\rm syn}} - I_i, \\ I_{\rm Na}(t) &= g_{\rm Na} m_i(t)^3 h_i(t)(V_i(t) - E_{\rm Na}) \\ I_{\rm K}(t) &= g_{\rm K} n_i(t)^4 (V_i(t) - E_{\rm K}) \\ \frac{dy(t)}{dt} &= \alpha_y(V(t))(1 - y(t)) - \beta_y(V(t))y(t), \end{aligned}$$

where $y_i = m, h, n$, and

$$\begin{aligned} \alpha_n &= 0.032(-50 - V)/\big(\exp((-50 - V)/5) - 1\big) \\ \beta_n &= 0.5\exp((-55 - V)/40) \\ \alpha_m &= 0.32(-52 - V)/\big(\exp((-52 - V)/4) - 1\big) \\ \beta_m &= 0.28(25 + V)/\big(\exp((25 + V)/5) - 1\big) \\ \alpha_h &= 0.128\exp((-48 - V)/18) \\ \beta_h &= 4/\big(\exp((-25 - V)/5) + 1\big). \end{aligned}$$

and typical parameters are $C = 0.143$ nF, $g_{\rm leak} = 0.02672$ μS, $E_{\rm leak} = -63.563$ mV, $g_{\rm Na} = 7.15$ μS, $E_{\rm Na} = 50$ mV, $g_{\rm K} = 1.43$ μS, $E_{\rm K} = -95$ 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

Note

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.

18.44.2 Member Typedef Documentation

18.44.2.1 typedef Snippet::ValueBase< 7 > NeuronModels::TraubMiles::ParamValues

18.44.2.2 typedef NewModels::VarInitContainerBase< 4 > NeuronModels::TraubMiles::VarValues

18.44.3 Member Function Documentation

18.44.3.1 static const NeuronModels::TraubMiles∗ NeuronModels::TraubMiles::getInstance ( ) `[inline]`, `[static]`

18.44.3.2 virtual StringVec NeuronModels::TraubMiles::getParamNames ( ) const `[inline]`, `[virtual]`

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

Reimplemented in NeuronModels::TraubMilesNStep.

18.44.3.3 virtual std::string NeuronModels::TraubMiles::getSimCode ( ) const `[inline]`, `[virtual]`

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.

Reimplemented from NeuronModels::Base.

Reimplemented in NeuronModels::TraubMilesNStep, NeuronModels::TraubMilesAlt, and NeuronModels::TraubMilesFast.

18.44.3.4 virtual std::string NeuronModels::TraubMiles::getThresholdConditionCode ( ) const `[inline]`, `[virtual]`

Gets code which defines the condition for a true spike in the described neuron model.

This evaluates to a bool (e.g. "V > 20").

Reimplemented from NeuronModels::Base.

18.44.3.5 virtual StringPairVec NeuronModels::TraubMiles::getVars ( ) const `[inline]`, `[virtual]`

Gets names and types (as strings) of model variables.

Reimplemented from NewModels::Base.

The documentation for this class was generated from the following file:

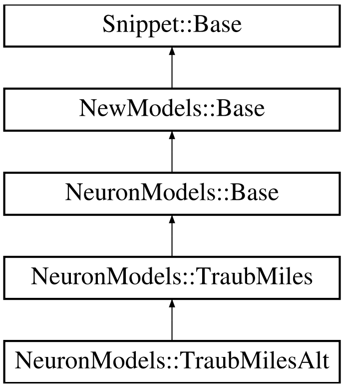
- newNeuronModels.h

18.45 NeuronModels::TraubMilesAlt Class Reference

Hodgkin-Huxley neurons with Traub & Miles algorithm.

```
#include <newNeuronModels.h>
```

Inheritance diagram for NeuronModels::TraubMilesAlt:



Public Types

- typedef Snippet::ValueBase< 7 > ParamValues
- typedef NewModels::VarInitContainerBase< 4 > VarValues

Public Member Functions

- virtual std::string getSimCode () const

  Gets the code that defines the execution of one timestep of integration of the neuron model.

Static Public Member Functions

- static const NeuronModels::TraubMilesAlt ∗ getInstance ()

18.45.1 Detailed Description

Hodgkin-Huxley neurons with Traub & Miles algorithm.

Using a workaround to avoid singularity: adding the munimum numerical value of the floating point precision used.

18.45.2 Member Typedef Documentation

18.45.2.1 typedef **Snippet::ValueBase< 7 > NeuronModels::TraubMilesAlt::ParamValues**

18.45.2.2 typedef **NewModels::VarInitContainerBase< 4 > NeuronModels::TraubMilesAlt::VarValues**

18.45.3 Member Function Documentation

18.45.3.1 static const **NeuronModels::TraubMilesAlt**∗ NeuronModels::TraubMilesAlt::getInstance ( ) `[inline]`, `[static]`

18.45.3.2 virtual std::string NeuronModels::TraubMilesAlt::getSimCode ( ) const `[inline]`, `[virtual]`

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.

Reimplemented from NeuronModels::TraubMiles.

The documentation for this class was generated from the following file:

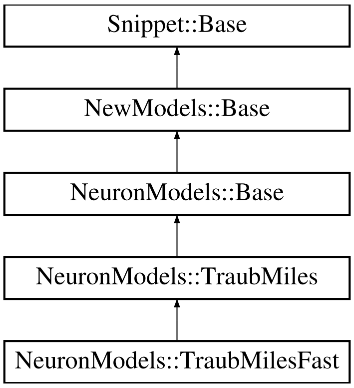
- newNeuronModels.h

18.46 NeuronModels::TraubMilesFast Class Reference

Hodgkin-Huxley neurons with Traub & Miles algorithm: Original fast implementation, using 25 inner iterations.

`#include <newNeuronModels.h>`

Inheritance diagram for NeuronModels::TraubMilesFast:



Public Types

- typedef Snippet::ValueBase< 7 > ParamValues
- typedef NewModels::VarInitContainerBase< 4 > VarValues

Public Member Functions

- virtual std::string getSimCode () const

  Gets the code that defines the execution of one timestep of integration of the neuron model.

Static Public Member Functions

- static const NeuronModels::TraubMilesFast ∗ getInstance ()

18.46.1 Detailed Description

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

18.46.2 Member Typedef Documentation

18.46.2.1 typedef **Snippet::ValueBase**< 7 > **NeuronModels::TraubMilesFast::ParamValues**

18.46.2.2 typedef **NewModels::VarInitContainerBase**< 4 > **NeuronModels::TraubMilesFast::VarValues**

18.46.3 Member Function Documentation

18.46.3.1 static const **NeuronModels::TraubMilesFast**∗ NeuronModels::TraubMilesFast::getInstance ( ) [inline], [static]

18.46.3.2 virtual std::string NeuronModels::TraubMilesFast::getSimCode ( ) const [inline], [virtual]

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.

Reimplemented from NeuronModels::TraubMiles.

The documentation for this class was generated from the following file:

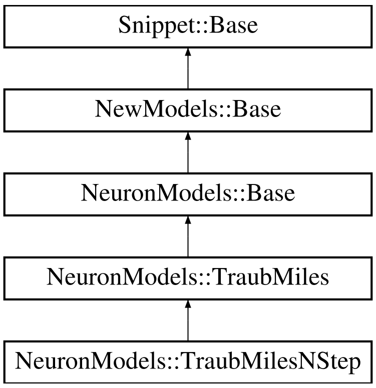
- newNeuronModels.h

18.47 NeuronModels::TraubMilesNStep Class Reference

Hodgkin-Huxley neurons with Traub & Miles algorithm.

```
#include <newNeuronModels.h>
```

Inheritance diagram for NeuronModels::TraubMilesNStep:



Public Types

- typedef Snippet::ValueBase< 8 > ParamValues
- typedef NewModels::VarInitContainerBase< 4 > VarValues

Public Member Functions

- virtual std::string getSimCode () const

  Gets the code that defines the execution of one timestep of integration of the neuron model.
- virtual StringVec getParamNames () const

  Gets names of of (independent) model parameters.

Static Public Member Functions

- static const NeuronModels::TraubMilesNStep ∗ getInstance ()

18.47.1 Detailed Description

Hodgkin-Huxley neurons with Traub & Miles algorithm.

Same as standard TraubMiles model but number of inner loops can be set using a parameter

18.47.2 Member Typedef Documentation

18.47.2.1 typedef Snippet::ValueBase< 8 > NeuronModels::TraubMilesNStep::ParamValues

18.47.2.2 typedef NewModels::VarInitContainerBase< 4 > NeuronModels::TraubMilesNStep::VarValues

18.47.3 Member Function Documentation

18.47.3.1 static const NeuronModels::TraubMilesNStep* NeuronModels::TraubMilesNStep::getInstance ( ) [inline], [static]

18.47.3.2 virtual StringVec NeuronModels::TraubMilesNStep::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from NeuronModels::TraubMiles.

18.47.3.3 virtual std::string NeuronModels::TraubMilesNStep::getSimCode ( ) const [inline], [virtual]

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.

Reimplemented from NeuronModels::TraubMiles.

The documentation for this class was generated from the following file:

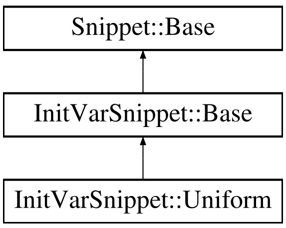
- newNeuronModels.h

18.48 InitVarSnippet::Uniform Class Reference

Initialises variable by sampling from the uniform distribution.

`#include <initVarSnippet.h>`

Inheritance diagram for InitVarSnippet::Uniform:



Public Member Functions

- DECLARE_SNIPPET (InitVarSnippet::Uniform, 2)
- SET_CODE ("const scalar scale = $(max) - $(min);\n""$(value) = $(min) + ($(gennrand_uniform) * scale);")
- virtual StringVec getParamNames () const

    Gets names of of (independent) model parameters.

Additional Inherited Members

18.48.1 Detailed Description

Initialises variable by sampling from the uniform distribution.

This snippet takes 2 parameters:

- `min` - The minimum value
- `max` - The maximum value

18.48.2 Member Function Documentation

18.48.2.1 InitVarSnippet::Uniform::DECLARE_SNIPPET (InitVarSnippet::Uniform , 2)

18.48.2.2 virtual StringVec InitVarSnippet::Uniform::getParamNames ( ) const [inline], [virtual]

Gets names of of (independent) model parameters.

Reimplemented from Snippet::Base.

18.48.2.3 InitVarSnippet::Uniform::SET_CODE ( )

The documentation for this class was generated from the following file:

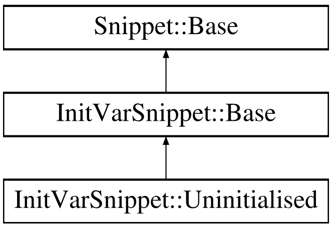
- initVarSnippet.h

18.49 InitVarSnippet::Uninitialised Class Reference

Used to mark variables as uninitialised - no initialisation code will be run.

`#include <initVarSnippet.h>`

Inheritance diagram for InitVarSnippet::Uninitialised:



Public Member Functions

- DECLARE_SNIPPET (InitVarSnippet::Uninitialised, 0)

Additional Inherited Members

18.49.1 Detailed Description

Used to mark variables as uninitialised - no initialisation code will be run.

18.49.2 Member Function Documentation

18.49.2.1 InitVarSnippet::Uninitialised::DECLARE_SNIPPET (InitVarSnippet::Uninitialised , 0)

The documentation for this class was generated from the following file:

- initVarSnippet.h

18.50 Snippet::ValueBase< NumVars > Class Template Reference

`#include <snippet.h>`

Public Member Functions

- template<typename... T>
  ValueBase (T &&...vals)
- const std::vector< double > & getValues () const

  Gets values as a vector of doubles.
- double operator[] (size_t pos) const

18.50.1 Constructor & Destructor Documentation

18.50.1.1 template<size_t NumVars> template<typename... T> Snippet::ValueBase< NumVars >::ValueBase (T &&... *vals*) [inline]

18.50.2 Member Function Documentation

18.50.2.1 template<size_t NumVars> const std::vector<double>& Snippet::ValueBase< NumVars >::getValues ( ) const [inline]

Gets values as a vector of doubles.

18.50.2.2 template<size_t NumVars> double Snippet::ValueBase< NumVars >::operator[] (size_t *pos*) const [inline]

The documentation for this class was generated from the following file:

- snippet.h

18.51 Snippet::ValueBase< 0 > Class Template Reference

`#include <snippet.h>`

Public Member Functions

- template<typename... T>
  ValueBase (T &&...vals)
- std::vector< double > getValues () const

  Gets values as a vector of doubles.

18.51.1 Detailed Description

template<>
class Snippet::ValueBase< 0 >

Template specialisation of ValueBase to avoid compiler warnings in the case when a model requires no parameters or state variables

18.51.2 Constructor & Destructor Documentation

18.51.2.1 template<typename... T> Snippet::ValueBase< 0 >::ValueBase (T &&... *vals*) [inline]

18.51.3 Member Function Documentation

18.51.3.1 std::vector<double> Snippet::ValueBase< 0 >::getValues ( ) const [inline]

Gets values as a vector of doubles.

The documentation for this class was generated from the following file:

- snippet.h

18.52 NewModels::VarInit Class Reference

```
#include <newModels.h>
```

Public Member Functions

- VarInit (const InitVarSnippet::Base ∗snippet, const std::vector< double > &params)
- VarInit (double constant)
- const InitVarSnippet::Base ∗ getSnippet () const
- const std::vector< double > & getParams () const
- const std::vector< double > & getDerivedParams () const
- void initDerivedParams (double dt)

18.52.1 Detailed Description

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

18.52.2 Constructor & Destructor Documentation

18.52.2.1 NewModels::VarInit::VarInit (const InitVarSnippet::Base ∗ *snippet*, const std::vector< double > & *params*) [inline]

18.52.2.2 NewModels::VarInit::VarInit (double *constant*) [inline]

18.52.3 Member Function Documentation

18.52.3.1 const std::vector<double>& NewModels::VarInit::getDerivedParams ( ) const [inline]

18.52.3.2 const std::vector<double>& NewModels::VarInit::getParams ( ) const [inline]

18.52.3.3 const InitVarSnippet::Base∗ NewModels::VarInit::getSnippet ( ) const [inline]

18.52.3.4 void NewModels::VarInit::initDerivedParams (double *dt*) [inline]

The documentation for this class was generated from the following file:

- newModels.h

18.53 NewModels::VarInitContainerBase< NumVars > Class Template Reference

`#include <newModels.h>`

Public Member Functions

- template<typename... T>
  VarInitContainerBase (T &&...initialisers)
- const std::vector< VarInit > & getInitialisers () const

  Gets initialisers as a vector of Values.
- const VarInit & operator[] (size_t pos) const

18.53.1 Detailed Description

template<size_t NumVars>
class NewModels::VarInitContainerBase< NumVars >

Wrapper to ensure at compile time that correct number of value initialisers are used when specifying the values of a model's initial state.

18.53.2 Constructor & Destructor Documentation

18.53.2.1 template<size_t NumVars> template<typename... T> NewModels::VarInitContainerBase< NumVars >::VarInitContainerBase (T &&... *initialisers*) [inline]

18.53.3 Member Function Documentation

18.53.3.1 template<size_t NumVars> const std::vector<VarInit>& NewModels::VarInitContainerBase< NumVars >::getInitialisers ( ) const [inline]

Gets initialisers as a vector of Values.

18.53.3.2 template<size_t NumVars> const VarInit& NewModels::VarInitContainerBase< NumVars >::operator[] (size_t *pos*) const [inline]

The documentation for this class was generated from the following file:

- newModels.h

18.54 NewModels::VarInitContainerBase< 0 > Class Template Reference

`#include <newModels.h>`

Public Member Functions

- template<typename... T>
  VarInitContainerBase (T &&...initialisers)
- VarInitContainerBase (const Snippet::ValueBase< 0 > &)
- std::vector< VarInit > getInitialisers () const

  Gets initialisers as a vector of Values.

18.54.1 Detailed Description

template<>
class NewModels::VarInitContainerBase< 0 >

Template specialisation of ValueInitBase to avoid compiler warnings in the case when a model requires no variable initialisers

18.54.2 Constructor & Destructor Documentation

18.54.2.1 template<typename... T> NewModels::VarInitContainerBase< 0 >::VarInitContainerBase (T &&... *initialisers*) [inline]

18.54.2.2 NewModels::VarInitContainerBase< 0 >::VarInitContainerBase (const Snippet::ValueBase< 0 > &) [inline]

18.54.3 Member Function Documentation

18.54.3.1 std::vector<VarInit> NewModels::VarInitContainerBase< 0 >::getInitialisers ( ) const [inline]

Gets initialisers as a vector of Values.

The documentation for this class was generated from the following file:

- newModels.h

18.55 weightUpdateModel Class Reference

Class to hold the information that defines a weightupdate model (a model of how spikes affect synaptic (and/or) (mostly) post-synaptic neuron variables. It also allows to define changes in response to post-synaptic spikes/spike-like events.

`#include <synapseModels.h>`

Public Member Functions

- weightUpdateModel ()
  Constructor for weightUpdateModel objects.
- ∼weightUpdateModel ()
  Destructor for weightUpdateModel objects.

Public Attributes

- string simCode
  Simulation code that is used for true spikes (only one time step after spike detection)
- string simCodeEvnt
  Simulation code that is used for spike events (all the instances where event threshold condition is met)
- string simLearnPost
  Simulation code which is used in the learnSynapsesPost kernel/function, where postsynaptic neuron spikes before the presynaptic neuron in the STDP window.
- string evntThreshold
  Simulation code for spike event detection.
- string synapseDynamics
  Simulation code for synapse dynamics independent of spike detection.

- string simCode_supportCode

  Support code is made available within the synapse kernel definition file and is meant 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; note that this support code is available to simCode, evntThreshold and simCodeEvnt.
- string simLearnPost_supportCode

  Support code is made available within the synapse kernel definition file and is meant 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.
- string synapseDynamics_supportCode

  Support code is made available within the synapse kernel definition file and is meant 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.
- vector< string > varNames

  Names of the variables in the postsynaptic model.
- vector< string > varTypes

  Types of the variable named above, e.g. "float". Names and types are matched by their order of occurrence in the vector.
- vector< string > pNames

  Names of (independent) parameters of the model.
- vector< string > dpNames

  Names of dependent parameters of the model.
- vector< string > extraGlobalSynapseKernelParameters

  Additional parameter in the neuron kernel; it is translated to a population specific name but otherwise assumed to be one parameter per population rather than per synapse.
- vector< string > extraGlobalSynapseKernelParameterTypes

  Additional parameters in the neuron kernel; they are translated to a population specific name but otherwise assumed to be one parameter per population rather than per synapse.
- dpclass * dps
- bool needPreSt

  Whether presynaptic spike times are needed or not.
- bool needPostSt

  Whether postsynaptic spike times are needed or not.

18.55.1 Detailed Description

Class to hold the information that defines a weightupdate model (a model of how spikes affect synaptic (and/or) (mostly) post-synaptic neuron variables. It also allows to define changes in response to post-synaptic spikes/spike-like events.

18.55.2 Constructor & Destructor Documentation

18.55.2.1 weightUpdateModel::weightUpdateModel ( )

Constructor for weightUpdateModel objects.

18.55.2.2 weightUpdateModel::∼weightUpdateModel ( )

Destructor for weightUpdateModel objects.

18.55.3 Member Data Documentation

18.55.3.1 vector<string> weightUpdateModel::dpNames

Names of dependent parameters of the model.

18.55.3.2 dpclass* weightUpdateModel::dps

18.55.3.3 string weightUpdateModel::evntThreshold

Simulation code for spike event detection.

18.55.3.4 vector<string> weightUpdateModel::extraGlobalSynapseKernelParameters

Additional parameter in the neuron kernel; it is translated to a population specific name but otherwise assumed to be one parameter per population rather than per synapse.

18.55.3.5 vector<string> weightUpdateModel::extraGlobalSynapseKernelParameterTypes

Additional parameters in the neuron kernel; they are translated to a population specific name but otherwise assumed to be one parameter per population rather than per synapse.

18.55.3.6 bool weightUpdateModel::needPostSt

Whether postsynaptic spike times are needed or not.

18.55.3.7 bool weightUpdateModel::needPreSt

Whether presynaptic spike times are needed or not.

18.55.3.8 vector<string> weightUpdateModel::pNames

Names of (independent) parameters of the model.

18.55.3.9 string weightUpdateModel::simCode

Simulation code that is used for true spikes (only one time step after spike detection)

18.55.3.10 string weightUpdateModel::simCode_supportCode

Support code is made available within the synapse kernel definition file and is meant 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; note that this support code is available to simCode, evntThreshold and simCodeEvnt.

18.55.3.11 string weightUpdateModel::simCodeEvnt

Simulation code that is used for spike events (all the instances where event threshold condition is met)

18.55.3.12 string weightUpdateModel::simLearnPost

Simulation code which is used in the learnSynapsesPost kernel/function, where postsynaptic neuron spikes before the presynaptic neuron in the STDP window.

18.55.3.13 string weightUpdateModel::simLearnPost_supportCode

Support code is made available within the synapse kernel definition file and is meant 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.

18.55.3.14 string weightUpdateModel::synapseDynamics

Simulation code for synapse dynamics independent of spike detection.

18.55.3.15 string weightUpdateModel::synapseDynamics_supportCode

Support code is made available within the synapse kernel definition file and is meant 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.

18.55.3.16 vector<string> weightUpdateModel::varNames

Names of the variables in the postsynaptic model.

18.55.3.17 vector<string> weightUpdateModel::varTypes

Types of the variable named above, e.g. "float". Names and types are matched by their order of occurrence in the vector.

The documentation for this class was generated from the following files:

- synapseModels.h
- synapseModels.cc

19 File Documentation

19.1 00_MainPage.dox File Reference

19.2 01_Installation.dox File Reference

19.3 02_Quickstart.dox File Reference

19.4 03_Examples.dox File Reference

19.5 05_SpineML.dox File Reference

19.6 06_Brian2GeNN.dox File Reference

19.7 09_ReleaseNotes.dox File Reference

19.8 10_UserManual.dox File Reference

19.9 11_Tutorial.dox File Reference

19.10 12_Tutorial.dox File Reference

19.11 13_UserGuide.dox File Reference

19.12 14_Credits.dox File Reference

19.13 codeGenUtils.cc File Reference

```
#include "codeGenUtils.h"
```

```
#include <regex>
#include "modelSpec.h"
#include "standardSubstitutions.h"
#include "utils.h"
```

Macros

- #define REGEX_OPERATIONAL

Enumerations

- enum MathsFunc

Functions

- void substitute (string &s, const string &trg, const string &rep)

  Tool for substituting strings in the neuron code strings or other templates.
- bool isRNGRequired (const std::string &code)

  Does the code string contain any functions requiring random number generator.
- bool isInitRNGRequired (const std::vector< NewModels::VarInit > &varInitialisers, const std::vector< VarMode > &varModes, VarInit initLocation)

  Does the model with the vectors of variable initialisers and modes require an RNG for the specified init mode.
- void functionSubstitute (std::string &code, const std::string &funcName, unsigned int numParams, const std::string &replaceFuncTemplate)

  This function substitutes function calls in the form:
- void functionSubstitutions (std::string &code, const std::string &ftype, const std::vector< FunctionTemplate > functions)

  This function performs a list of function substitutions in code snipped.
- string ensureFtype (const string &oldcode, const 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 string &code, const string &codeName)

  This function checks for unknown variable definitions and returns a gennError if any are found.
- void neuron_substitutions_in_synaptic_code (string &wCode, const SynapseGroup *sg, const string &preIdx, const string &postIdx, const string &devPrefix)

  Function for performing the code and value substitutions necessary to insert neuron related variables, parameters, and extraGlobal parameters into synaptic code.

19.13.1 Macro Definition Documentation

19.13.1.1 #define REGEX_OPERATIONAL

19.13.2 Enumeration Type Documentation

19.13.2.1 enum MathsFunc

19.13.3 Function Documentation

19.13.3.1 void checkUnreplacedVariables (const string & *code*, const string & *codeName*)

This function checks for unknown variable definitions and returns a gennError if any are found.

19.13.3.2 string ensureFtype (const string & *oldcode,* const 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).

19.13.3.3 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))

19.13.3.4 void functionSubstitutions (std::string & *code,* const std::string & *ftype,* const std::vector< FunctionTemplate > *functions*)

This function performs a list of function substitutions in code snipped.

19.13.3.5 bool isInitRNGRequired (const std::vector< NewModels::VarInit > & *varInitialisers,* const std::vector< VarMode > & *varModes,* VarInit *initLocation*)

Does the model with the vectors of variable initialisers and modes require an RNG for the specified init mode.

Does the model with the vectors of variable initialisers and modes require an RNG for the specified init location i.e. host or device.

19.13.3.6 bool isRNGRequired (const std::string & *code*)

Does the code string contain any functions requiring random number generator.

19.13.3.7 void neuron_substitutions_in_synaptic_code (string & *wCode,* const SynapseGroup ∗ *sg,* const string & *preIdx,* const string & *postIdx,* const string & *devPrefix*)

Function for performing the code and value substitutions necessary to insert neuron related variables, parameters, and extraGlobal parameters into synaptic code.

Parameters

wCode	the code string to work on
preIdx	index of the pre-synaptic neuron to be accessed for _pre variables; differs for different Span)
postIdx	index of the post-synaptic neuron to be accessed for _post variables; differs for different Span)
devPrefix	device prefix, "dd_" for GPU, nothing for CPU

19.13.3.8 void substitute (string & *s,* const string & *trg,* const string & *rep*)

Tool for substituting strings in the neuron code strings or other templates.

19.14 codeGenUtils.h File Reference

```
#include <limits>
#include <string>
#include <sstream>
#include <vector>
#include "variableMode.h"
```

Classes

- struct GenericFunction
- struct FunctionTemplate
- class PairKeyConstIter< BaseIter >

  Custom iterator for iterating through the keys of containers containing pairs.

Namespaces

- NeuronModels
- NewModels

Functions

- template<typename BaseIter >
  PairKeyConstIter< BaseIter > GetPairKeyConstIter (BaseIter iter)

  Helper function for creating a PairKeyConstIter from an iterator.
- void substitute (string &s, const string &trg, const string &rep)

  Tool for substituting strings in the neuron code strings or other templates.
- bool isRNGRequired (const std::string &code)

  Does the code string contain any functions requiring random number generator.
- bool isInitRNGRequired (const std::vector< NewModels::VarInit > &varInitialisers, const std::vector< VarMode > &varModes, VarInit initLocation)

  Does the model with the vectors of variable initialisers and modes require an RNG for the specified init location i.e. host or device.
- void functionSubstitute (std::string &code, const std::string &funcName, unsigned int numParams, const std::string &replaceFuncTemplate)

  This function substitutes function calls in the form:
- template<typename NameIter >
  void name_substitutions (string &code, const string &prefix, NameIter namesBegin, NameIter namesEnd, const string &postfix="", const string &ext="")

  This function performs a list of name substitutions for variables in code snippets.
- void name_substitutions (string &code, const string &prefix, const vector< string > &names, const string &postfix="", const string &ext="")

  This function performs a list of name substitutions for variables in code snippets.
- template<typename NameIter >
  void value_substitutions (string &code, NameIter namesBegin, NameIter namesEnd, const vector< double > &values, const string &ext="")

  This function performs a list of value substitutions for parameters in code snippets.
- void value_substitutions (string &code, const vector< string > &names, const vector< double > &values, const string &ext="")

  This function performs a list of value substitutions for parameters in code snippets.
- void functionSubstitutions (std::string &code, const std::string &ftype, const std::vector< FunctionTemplate > functions)

  This function performs a list of function substitutions in code snipped.
- string ensureFtype (const string &oldcode, const 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 string &code, const string &codeName)

  This function checks for unknown variable definitions and returns a gennError if any are found.

Variables

- const std::vector< FunctionTemplate > cudaFunctions

  CUDA implementations of standard functions.
- const std::vector< FunctionTemplate > cpuFunctions

  CPU implementations of standard functions.

19.14.1 Function Documentation

19.14.1.1 void checkUnreplacedVariables (const string & *code,* const string & *codeName*)

This function checks for unknown variable definitions and returns a gennError if any are found.

19.14.1.2 string ensureFtype (const string & *oldcode,* const 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).

19.14.1.3 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))

19.14.1.4 void functionSubstitutions (std::string & *code,* const std::string & *ftype,* const std::vector< FunctionTemplate > *functions*)

This function performs a list of function substitutions in code snipped.

19.14.1.5 template<typename BaseIter > PairKeyConstIter<BaseIter> GetPairKeyConstIter (BaseIter *iter*) [inline]

Helper function for creating a PairKeyConstIter from an iterator.

19.14.1.6 bool isInitRNGRequired (const std::vector< NewModels::VarInit > & *varInitialisers,* const std::vector< VarMode > & *varModes,* VarInit *initLocation*)

Does the model with the vectors of variable initialisers and modes require an RNG for the specified init location i.e. host or device.

Does the model with the vectors of variable initialisers and modes require an RNG for the specified init location i.e. host or device.

19.14.1.7 bool isRNGRequired (const std::string & *code*)

Does the code string contain any functions requiring random number generator.

19.14.1.8 template<typename NameIter > void name_substitutions (string & *code,* const string & *prefix,* NameIter *namesBegin,* NameIter *namesEnd,* const string & *postfix* = "", const string & *ext* = "") [inline]

This function performs a list of name substitutions for variables in code snippets.

19.14.1.9 void name_substitutions (string & *code,* const string & *prefix,* const vector< string > & *names,* const string & *postfix* = "", const string & *ext* = "") [inline]

This function performs a list of name substitutions for variables in code snippets.

19.14.1.10 void substitute (string & *s*, const string & *trg*, const string & *rep*)

Tool for substituting strings in the neuron code strings or other templates.

19.14.1.11 template<typename NameIter > void value_substitutions (string & *code*, NameIter *namesBegin*, NameIter *namesEnd*, const vector< double > & *values*, const string & *ext* = "") [inline]

This function performs a list of value substitutions for parameters in code snippets.

19.14.1.12 void value_substitutions (string & *code*, const vector< string > & *names*, const vector< double > & *values*, const string & *ext* = "") [inline]

This function performs a list of value substitutions for parameters in code snippets.

19.14.2 Variable Documentation

19.14.2.1 const std::vector<FunctionTemplate> cpuFunctions

Initial value:

```
= {
    {"gennrand_uniform", 0, "standardUniformDistribution($(rng))", "standardUniformDistribution($(rng))"},
    {"gennrand_normal", 0, "standardNormalDistribution($(rng))", "standardNormalDistribution($(rng))"},
    {"gennrand_exponential", 0, "standardExponentialDistribution($(rng))", "
      standardExponentialDistribution($(rng))"},
    {"gennrand_log_normal", 2, "std::lognormal_distribution<double>($(0), $(1))($(rng))", "
      std::lognormal_distribution<float>($(0), $(1))($(rng))"},
}
```

CPU implementations of standard functions.

19.14.2.2 const std::vector<FunctionTemplate> cudaFunctions

Initial value:

```
= {
    {"gennrand_uniform", 0, "curand_uniform_double($(rng))", "curand_uniform($(rng))"},
    {"gennrand_normal", 0, "curand_normal_double($(rng))", "curand_normal($(rng))"},
    {"gennrand_exponential", 0, "exponentialDistFloat($(rng))", "exponentialDistDouble($(rng))"},
    {"gennrand_log_normal", 2, "curand_log_normal_double($(rng), $(0), $(1))", "
      curand_log_normal_float($(rng), $(0), $(1))"},
}
```

CUDA implementations of standard functions.

19.15 codeStream.cc File Reference

```
#include "codeStream.h"
#include <algorithm>
#include "utils.h"
```

Functions

- std::ostream & operator<< (std::ostream &s, const CodeStream::OB &ob)
- std::ostream & operator<< (std::ostream &s, const CodeStream::CB &cb)

19.15.1 Function Documentation

19.15.1.1 std::ostream& operator<< (std::ostream & *s*, const CodeStream::OB & *ob*)

19.15.1.2 std::ostream& operator<< (std::ostream & *s*, const CodeStream::CB & *cb*)

19.16 codeStream.h File Reference

```
#include <ostream>
#include <streambuf>
#include <string>
#include <vector>
```

Classes

- class CodeStream

  Helper class for generating code - automatically inserts brackets, indents etc.
- struct CodeStream::OB

  An open bracket marker.
- struct CodeStream::CB

  A close bracket marker.

Functions

- std::ostream & operator<< (std::ostream &s, const CodeStream::OB &ob)
- std::ostream & operator<< (std::ostream &s, const CodeStream::CB &cb)

19.16.1 Function Documentation

19.16.1.1 std::ostream& operator<< (std::ostream & *s*, const CodeStream::OB & *ob*)

19.16.1.2 std::ostream& operator<< (std::ostream & *s*, const CodeStream::CB & *cb*)

19.17 dpclass.h File Reference

```
#include <vector>
```

Classes

- class dpclass

19.18 extra_neurons.h File Reference

Functions

- n varNames clear ()
- n varNames push_back ("V")
- n varTypes push_back ("float")
- n varNames push_back ("V_NB")
- n varNames push_back ("tSpike_NB")
- n varNames push_back ("__regime_val")
- n varTypes push_back ("int")
- n pNames push_back ("VReset_NB")
- n pNames push_back ("VThresh_NB")
- n pNames push_back ("tRefrac_NB")

- n pNames push_back ("VRest_NB")
- n pNames push_back ("TAUm_NB")
- n pNames push_back ("Cm_NB")
- nModels push_back (n)
- n varNames push_back ("count_t_NB")
- n pNames push_back ("max_t_NB")

Variables

- n simCode

19.18.1 Function Documentation

19.18.1.1 ps dpNames clear ( )

19.18.1.2 n varNames push_back ("V")

19.18.1.3 ps varTypes push_back ("float")

19.18.1.4 n varNames push_back ("V_NB")

19.18.1.5 n varNames push_back ("tSpike_NB")

19.18.1.6 n varNames push_back ("__regime_val")

19.18.1.7 n varTypes push_back ("int")

19.18.1.8 n pNames push_back ("VReset_NB")

19.18.1.9 n pNames push_back ("VThresh_NB")

19.18.1.10 n pNames push_back ("tRefrac_NB")

19.18.1.11 n pNames push_back ("VRest_NB")

19.18.1.12 n pNames push_back ("TAUm_NB")

19.18.1.13 n pNames push_back ("Cm_NB")

19.18.1.14 nModels push_back (n)

19.18.1.15 n varNames push_back ("count_t_NB")

19.18.1.16 n pNames push_back ("max_t_NB")

19.18.2 Variable Documentation

19.18.2.1 n simCode

Initial value:

```
= " \
            $(V) = -1000000; \
            if ($(__regime_val)==1) { \n \
$(V_NB) += (Isyn_NB/$(Cm_NB)+($(VRest_NB)-$(V_NB))/$(TAUm_NB))*DT; \n \
                   if ($(V_NB)>$(VThresh_NB)) { \n \
$(V_NB) = $(VReset_NB); \n \
$(tSpike_NB) = t; \n \
                  $(V) = 100000; \
$(__regime_val) = 2; \n \
} \n \
} \n \
if ($(__regime_val)==2) { \n \
```

```
if (t-$(tSpike_NB) > $(tRefrac_NB)) { \n \
$(__regime_val) = 1; \n \
} \n \
} \n \
"
```

19.19 extra_postsynapses.h File Reference

Functions

- ps varNames clear ()
- ps varNames push_back ("g_PS")
- ps varTypes push_back ("float")
- ps pNames push_back ("tau_syn_PS")
- ps pNames push_back ("E_PS")
- postSynModels push_back (ps)

Variables

- ps postSyntoCurrent
- ps postSynDecay

19.19.1 Function Documentation

19.19.1.1 ps varNames clear ( )

19.19.1.2 ps varNames push_back ("g_PS")

19.19.1.3 ps varTypes push_back ("float")

19.19.1.4 ps pNames push_back ("tau_syn_PS")

19.19.1.5 ps pNames push_back ("E_PS")

19.19.1.6 postSynModels push_back (ps)

19.19.2 Variable Documentation

19.19.2.1 ps postSynDecay

Initial value:

```
= " \
           $(g_PS) += (-$(g_PS)/$(tau_syn_PS))*DT; \n \
                          $(inSyn) = 0; \
          "
```

19.19.2.2 ps postSyntoCurrent

Initial value:

```
= " \
  0; \n \
             float Isyn_NB = 0; \n \
              { \n \
            float v_PS = lV_NB; \n \
              float g_in_PS = $(inSyn); \
$(g_PS) = $(g_PS)+g_in_PS; \n \
Isyn_NB += ($(g_PS)*($(E_PS)-v_PS)); \n \
          } \n \
"
```

19.20 extra_weightupdates.h File Reference

19.21 generateALL.cc File Reference

Main file combining the code for code generation. Part of the code generation section.

```
#include "global.h"
#include "generateALL.h"
#include "generateCPU.h"
#include "generateInit.h"
#include "generateKernels.h"
#include "generateRunner.h"
#include "modelSpec.h"
#include "utils.h"
#include "codeGenUtils.h"
#include "codeStream.h"
#include <algorithm>
#include <cmath>
#include <iterator>
#include <sys/stat.h>
#include <MODEL>
```

Functions

- void generate_model_runner (const NNmodel &model, const string &path)

  This function will call the necessary sub-functions to generate the code for simulating a model.
- void chooseDevice (NNmodel &model, const string &path)

  Helper function that prepares data structures and detects the hardware properties to enable the code generation code that follows.
- int main (int argc, char ∗argv[])

  Main entry point for the generateALL executable that generates the code for GPU and CPU.

19.21.1 Detailed Description

Main file combining the code for code generation. Part of the code generation section.

The file includes separate files for generating kernels (generateKernels.cc), generating the CPU side code for running simulations on either the CPU or GPU (generateRunner.cc) and for CPU-only simulation code (generateCP←U.cc).

19.21.2 Function Documentation

19.21.2.1 void chooseDevice (NNmodel & *model,* const string & *path*)

Helper function that prepares data structures and detects the hardware properties to enable the code generation code that follows.

The main tasks in this function are the detection and characterization of the GPU device present (if any), choosing which GPU device to use, finding and appropriate block size, taking note of the major and minor version of the C←UDA enabled device chosen for use, and populating the list of standard neuron models. The chosen device number is returned.

Parameters

model	the nn model we are generating code for
path	path the generated code will be deposited

19.21.2.2 void generate_model_runner (const NNmodel & *model*, const string & *path*)

This function will call the necessary sub-functions to generate the code for simulating a model.

Parameters

model	Model description
path	Path where the generated code will be deposited

19.21.2.3 int main (int *argc*, char * *argv[]*)

Main entry point for the generateALL executable that generates the code for GPU and CPU.

The main function is the entry point for the code generation engine. It prepares the system and then invokes generate_model_runner to inititate the different parts of actual code generation.

Parameters

argc	number of arguments; expected to be 2
argv	Arguments; expected to contain the target directory for code generation.

19.22 generateALL.h File Reference

```
#include "modelSpec.h"
#include <string>
```

Functions

- void generate_model_runner (const NNmodel &model, const string &path)

  This function will call the necessary sub-functions to generate the code for simulating a model.
- void chooseDevice (NNmodel &model, const string &path)

  Helper function that prepares data structures and detects the hardware properties to enable the code generation code that follows.

19.22.1 Function Documentation

19.22.1.1 void chooseDevice (NNmodel & *model*, const string & *path*)

Helper function that prepares data structures and detects the hardware properties to enable the code generation code that follows.

The main tasks in this function are the detection and characterization of the GPU device present (if any), choosing which GPU device to use, finding and appropriate block size, taking note of the major and minor version of the C↩ UDA enabled device chosen for use, and populating the list of standard neuron models. The chosen device number is returned.

Parameters

model	the nn model we are generating code for
path	path the generated code will be deposited

19.22.1.2 void generate_model_runner (const NNmodel & *model*, const string & *path*)

This function will call the necessary sub-functions to generate the code for simulating a model.

Parameters

model	Model description
path	Path where the generated code will be deposited

19.23 generateCPU.cc File Reference

Functions for generating code that will run the neuron and synapse simulations on the CPU. Part of the code generation section.

```
#include "generateCPU.h"
#include "global.h"
#include "utils.h"
#include "codeGenUtils.h"
#include "standardGeneratedSections.h"
#include "standardSubstitutions.h"
#include "codeStream.h"
#include <algorithm>
#include <typeinfo>
```

Functions

- void genNeuronFunction (const NNmodel &model, const string &path)

    Function that generates the code of the function the will simulate all neurons on the CPU.
- void genSynapseFunction (const NNmodel &model, const string &path)

    Function that generates code that will simulate all synapses of the model on the CPU.

19.23.1 Detailed Description

Functions for generating code that will run the neuron and synapse simulations on the CPU. Part of the code generation section.

19.23.2 Function Documentation

19.23.2.1 void genNeuronFunction (const NNmodel & *model*, const string & *path*)

Function that generates the code of the function the will simulate all neurons on the CPU.

Parameters

model	Model description
path	Path for code generation

19.23.2.2 void genSynapseFunction (const NNmodel & *model*, const string & *path*)

Function that generates code that will simulate all synapses of the model on the CPU.

Parameters

model	Model description
path	Path for code generation

19.24 generateCPU.h File Reference

Functions for generating code that will run the neuron and synapse simulations on the CPU. Part of the code generation section.

```
#include "modelSpec.h"
#include <string>
#include <fstream>
```

Functions

- void genNeuronFunction (const NNmodel &model, const string &path)

  Function that generates the code of the function the will simulate all neurons on the CPU.
- void genSynapseFunction (const NNmodel &model, const string &path)

  Function that generates code that will simulate all synapses of the model on the CPU.

19.24.1 Detailed Description

Functions for generating code that will run the neuron and synapse simulations on the CPU. Part of the code generation section.

19.24.2 Function Documentation

19.24.2.1 void genNeuronFunction (const **NNmodel** & *model*, const string & *path*)

Function that generates the code of the function the will simulate all neurons on the CPU.

Parameters

model	Model description
path	Path for code generation

19.24.2.2 void genSynapseFunction (const **NNmodel** & *model*, const string & *path*)

Function that generates code that will simulate all synapses of the model on the CPU.

Parameters

model	Model description
path	Path for code generation

19.25 generateInit.cc File Reference

```
#include "generateInit.h"
#include <algorithm>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include "codeStream.h"
#include "global.h"
#include "modelSpec.h"
#include "standardSubstitutions.h"
```

Functions

- void genInit (const NNmodel &model, const std::string &path)
  Path for code generationn.

19.25.1 Function Documentation

19.25.1.1 void genInit (const NNmodel & *model*, const std::string & *path*)

Path for code generationn.

Parameters

model	Model description
path	Path for code generationn

19.26 generateInit.h File Reference

Contains functions to generate code for initialising kernel state variables. Part of the code generation section.

```
#include <string>
```

Functions

- void genInit (const NNmodel &model, const std::string &path)
  Path for code generationn.

19.26.1 Detailed Description

Contains functions to generate code for initialising kernel state variables. Part of the code generation section.

19.26.2 Function Documentation

19.26.2.1 void genInit (const NNmodel & *model*, const std::string & *path*)

Path for code generationn.

Parameters

model	Model description
path	Path for code generationn

19.27 generateKernels.cc File Reference

Contains functions that generate code for CUDA kernels. Part of the code generation section.

```
#include "generateKernels.h"
#include "global.h"
#include "utils.h"
#include "standardGeneratedSections.h"
#include "standardSubstitutions.h"
#include "codeGenUtils.h"
#include "codeStream.h"
#include <algorithm>
```

Functions

- void genNeuronKernel (const NNmodel &model, const string &path)
  Function for generating the CUDA kernel that simulates all neurons in the model.
- void genSynapseKernel (const NNmodel &model, const string &path)
  Function for generating a CUDA kernel for simulating all synapses.

19.27.1 Detailed Description

Contains functions that generate code for CUDA kernels. Part of the code generation section.

19.27.2 Function Documentation

19.27.2.1 void genNeuronKernel (const NNmodel & *model*, const string & *path*)

Function for generating the CUDA kernel that simulates all neurons in the model.

The code generated upon execution of this function is for defining GPU side global variables that will hold model state in the GPU global memory and for the actual kernel function for simulating the neurons for one time step.

Parameters

model	Model description
path	Path for code generation

19.27.2.2 void genSynapseKernel (const NNmodel & *model*, const string & *path*)

Function for generating a CUDA kernel for simulating all synapses.

This functions generates code for global variables on the GPU side that are synapse-related and the actual CUDA kernel for simulating one time step of the synapses. $<$ "id" if first synapse group, else "lid". lid =(thread index- last thread of the last synapse group)

Parameters

model	Model description
path	Path for code generation

19.28 generateKernels.h File Reference

Contains functions that generate code for CUDA kernels. Part of the code generation section.

```
#include "modelSpec.h"
#include <string>
#include <fstream>
```

Functions

- void genNeuronKernel (const NNmodel &model, const string &path)

  Function for generating the CUDA kernel that simulates all neurons in the model.
- void genSynapseKernel (const NNmodel &model, const string &path)

  Function for generating a CUDA kernel for simulating all synapses.

19.28.1 Detailed Description

Contains functions that generate code for CUDA kernels. Part of the code generation section.

19.28.2 Function Documentation

19.28.2.1 void genNeuronKernel (const NNmodel & *model*, const string & *path*)

Function for generating the CUDA kernel that simulates all neurons in the model.

The code generated upon execution of this function is for defining GPU side global variables that will hold model state in the GPU global memory and for the actual kernel function for simulating the neurons for one time step.

Parameters

model	Model description
path	Path for code generation

19.28.2.2 void genSynapseKernel (const NNmodel & *model*, const string & *path*)

Function for generating a CUDA kernel for simulating all synapses.

This functions generates code for global variables on the GPU side that are synapse-related and the actual CUDA kernel for simulating one time step of the synapses. < "id" if first synapse group, else "lid". lid =(thread index- last thread of the last synapse group)

Parameters

model	Model description
path	Path for code generation

19.29 generateRunner.cc File Reference

Contains functions to generate code for running the simulation on the GPU, and for I/O convenience functions between GPU and CPU space. Part of the code generation section.

```
#include "generateRunner.h"
#include "global.h"
#include "utils.h"
#include "codeGenUtils.h"
#include "codeStream.h"
#include <algorithm>
#include <cfloat>
#include <cstdint>
```

Functions

- void genDefinitions (const NNmodel &model, const string &path)

  A function that generates predominantly host-side code.
- void genSupportCode (const NNmodel &model, const string &path)

  Path for code generationn.
- void genRunner (const NNmodel &model, const string &path)

  Path for code generationn.
- void genRunnerGPU (const NNmodel &model, const string &path)

  A function to generate the code that simulates the model on the GPU.
- void genMSBuild (const NNmodel &model, const string &path)

  A function that generates an MSBuild script for all generated GeNN code.
- void genMakefile (const NNmodel &model, const string &path)

  A function that generates the Makefile for all generated GeNN code.

19.29.1 Detailed Description

Contains functions to generate code for running the simulation on the GPU, and for I/O convenience functions between GPU and CPU space. Part of the code generation section.

19.29.2 Function Documentation

19.29.2.1 void genDefinitions (const NNmodel & *model,* const string & *path*)

A function that generates predominantly host-side code.

In this function host-side functions and other code are generated, including: Global host variables, "allocatedMem()" function for allocating memories, "freeMem" function for freeing the allocated memories, "initialize" for initializing host variables, "gFunc" and "initGRaw()" for use with plastic synapses if such synapses exist in the model.

Parameters

model	Model description
path	Path for code generationn

19.29.2.2 void genMakefile (const NNmodel & *model,* const string & *path*)

A function that generates the Makefile for all generated GeNN code.

Path for code generation

Parameters

model	Model description
path	Path for code generation

19.29.2.3 void genMSBuild (const NNmodel & *model,* const string & *path*)

A function that generates an MSBuild script for all generated GeNN code.

Path for code generation

Parameters

model	Model description
path	Path for code generation

19.29.2.4 void genRunner (const NNmodel & *model,* const string & *path*)

Path for code generationn.

Method for cleaning up and resetting device while quitting GeNN

Parameters

model	Model description
path	Path for code generationn

19.29.2.5 void genRunnerGPU (const NNmodel & *model,* const string & *path*)

A function to generate the code that simulates the model on the GPU.

The function generates functions that will spawn kernel grids onto the GPU (but not the actual kernel code which is generated in "genNeuronKernel()" and "genSynpaseKernel()"). Generated functions include "copyGToDevice()", "copyGFromDevice()", "copyStateToDevice()", "copyStateFromDevice()", "copySpikesFromDevice()", "copySpikeNFromDevice()" and "stepTimeGPU()". The last mentioned function is the function that will initialize the execution on the GPU in the generated simulation engine. All other generated functions are "convenience functions" to handle data transfer from and to the GPU.

Parameters

model	Model description
path	Path for code generation

19.29.2.6 void genSupportCode (const NNmodel & *model,* const string & *path*)

Path for code generationn.

Parameters

model	Model description
path	Path for code generationn

19.30 generateRunner.h File Reference

Contains functions to generate code for running the simulation on the GPU, and for I/O convenience functions between GPU and CPU space. Part of the code generation section.

```
#include "modelSpec.h"
#include <string>
#include <fstream>
```

Functions

- void genDefinitions (const NNmodel &model, const string &path)

    A function that generates predominantly host-side code.
- void genRunner (const NNmodel &model, const string &path)

    Path for code generationn.
- void genSupportCode (const NNmodel &model, const string &path)

    Path for code generationn.
- void genRunnerGPU (const NNmodel &model, const string &path)

    A function to generate the code that simulates the model on the GPU.
- void genMSBuild (const NNmodel &model, const string &path)

    A function that generates an MSBuild script for all generated GeNN code.
- void genMakefile (const NNmodel &model, const string &path)

    A function that generates the Makefile for all generated GeNN code.

19.30.1 Detailed Description

Contains functions to generate code for running the simulation on the GPU, and for I/O convenience functions between GPU and CPU space. Part of the code generation section.

19.30.2 Function Documentation

19.30.2.1 void genDefinitions (const **NNmodel** & *model,* const string & *path*)

A function that generates predominantly host-side code.

In this function host-side functions and other code are generated, including: Global host variables, "allocatedMem()" function for allocating memories, "freeMem" function for freeing the allocated memories, "initialize" for initializing host variables, "gFunc" and "initGRaw()" for use with plastic synapses if such synapses exist in the model. Path for code generationn

In this function host-side functions and other code are generated, including: Global host variables, "allocatedMem()" function for allocating memories, "freeMem" function for freeing the allocated memories, "initialize" for initializing host variables, "gFunc" and "initGRaw()" for use with plastic synapses if such synapses exist in the model.

Parameters

model	Model description
path	Path for code generationn

19.30.2.2 void genMakefile (const **NNmodel** & *model,* const string & *path*)

A function that generates the Makefile for all generated GeNN code.

Path for code generation

Parameters

model	Model description
path	Path for code generation

19.30.2.3 void genMSBuild (const NNmodel & *model,* const string & *path*)

A function that generates an MSBuild script for all generated GeNN code.

Path for code generation

Parameters

model	Model description
path	Path for code generation

19.30.2.4 void genRunner (const NNmodel & *model,* const string & *path*)

Path for code generationn.

Method for cleaning up and resetting device while quitting GeNN

Parameters

model	Model description
path	Path for code generationn

19.30.2.5 void genRunnerGPU (const NNmodel & *model,* const string & *path*)

A function to generate the code that simulates the model on the GPU.

The function generates functions that will spawn kernel grids onto the GPU (but not the actual kernel code which is generated in "genNeuronKernel()" and "genSynpaseKernel()"). Generated functions include "copyGToDevice()", "copyGFromDevice()", "copyStateToDevice()", "copyStateFromDevice()", "copySpikesFromDevice()", "copySpikeNFromDevice()" and "stepTimeGPU()". The last mentioned function is the function that will initialize the execution on the GPU in the generated simulation engine. All other generated functions are "convenience functions" to handle data transfer from and to the GPU.Path for code generation

The function generates functions that will spawn kernel grids onto the GPU (but not the actual kernel code which is generated in "genNeuronKernel()" and "genSynpaseKernel()"). Generated functions include "copyGToDevice()", "copyGFromDevice()", "copyStateToDevice()", "copyStateFromDevice()", "copySpikesFromDevice()", "copySpikeNFromDevice()" and "stepTimeGPU()". The last mentioned function is the function that will initialize the execution on the GPU in the generated simulation engine. All other generated functions are "convenience functions" to handle data transfer from and to the GPU.

Parameters

model	Model description
path	Path for code generation

19.30.2.6 void genSupportCode (const NNmodel & *model,* const string & *path*)

Path for code generationn.

Parameters

model	Model description
path	Path for code generationn

19.31 global.cc File Reference

```
#include "global.h"
```

Namespaces

- GENN_FLAGS
- GENN_PREFERENCES

Macros

- #define GLOBAL_CC

Variables

- unsigned int neuronBlkSz

    Global variable containing the GPU block size for the neuron kernel.
- unsigned int synapseBlkSz

    Global variable containing the GPU block size for the synapse kernel.
- unsigned int learnBlkSz

    Global variable containing the GPU block size for the learn kernel.
- unsigned int synDynBlkSz

    Global variable containing the GPU block size for the synapse dynamics kernel.
- unsigned int initBlkSz

    Global variable containing the GPU block size for the initialization kernel.
- unsigned int initSparseBlkSz

    Global variable containing the GPU block size for the sparse initialization kernel.
- cudaDeviceProp * deviceProp
- int theDevice

    Global variable containing the currently selected CUDA device's number.
- int deviceCount

    Global variable containing the number of CUDA devices on this host.
- int hostCount

    Global variable containing the number of hosts within the local compute cluster.

19.31.1 Macro Definition Documentation

19.31.1.1 #define GLOBAL_CC

19.31.2 Variable Documentation

19.31.2.1 int deviceCount

Global variable containing the number of CUDA devices on this host.

19.31.2.2 cudaDeviceProp* deviceProp

19.31.2.3 int hostCount

Global variable containing the number of hosts within the local compute cluster.

19.31.2.4 unsigned int initBlkSz

Global variable containing the GPU block size for the initialization kernel.

19.31.2.5 unsigned int initSparseBlkSz

Global variable containing the GPU block size for the sparse initialization kernel.

19.31.2.6 unsigned int learnBlkSz

Global variable containing the GPU block size for the learn kernel.

19.31.2.7 unsigned int neuronBlkSz

Global variable containing the GPU block size for the neuron kernel.

19.31.2.8 unsigned int synapseBlkSz

Global variable containing the GPU block size for the synapse kernel.

19.31.2.9 unsigned int synDynBlkSz

Global variable containing the GPU block size for the synapse dynamics kernel.

19.31.2.10 int theDevice

Global variable containing the currently selected CUDA device's number.

19.32 global.h File Reference

Global header file containing a few global variables. Part of the code generation section.

```
#include <string>
#include <cuda.h>
#include <cuda_runtime.h>
#include "variableMode.h"
```

Namespaces

- GENN_FLAGS
- GENN_PREFERENCES

Variables

- const unsigned int GENN_FLAGS::calcSynapseDynamics = 0
- const unsigned int GENN_FLAGS::calcSynapses = 1
- const unsigned int GENN_FLAGS::learnSynapsesPost = 2
- const unsigned int GENN_FLAGS::calcNeurons = 3
- bool GENN_PREFERENCES::optimiseBlockSize = true

  Flag for signalling whether or not block size optimisation should be performed.
- bool GENN_PREFERENCES::autoChooseDevice = true

Flag to signal whether the GPU device should be chosen automatically.

- bool GENN_PREFERENCES::optimizeCode = false

  Request speed-optimized code, at the expense of floating-point accuracy.
- bool GENN_PREFERENCES::debugCode = false

  Request debug data to be embedded in the generated code.
- bool GENN_PREFERENCES::showPtxInfo = false

  Request that PTX assembler information be displayed for each CUDA kernel during compilation.
- bool GENN_PREFERENCES::buildSharedLibrary = false

  Should generated code and Makefile build into a shared library e.g. for use in SpineML simulator.
- bool GENN_PREFERENCES::autoInitSparseVars = false

  Previously, variables associated with sparse synapse populations were not automatically initialised. If this flag is set this now occurs in the initMODEL_NAME function and copyStateToDevice is deferred until here.
- VarMode GENN_PREFERENCES::defaultVarMode = VarMode::LOC_HOST_DEVICE_INIT_HOST

  What is the default behaviour for model state variables? Historically, everything was allocated on both host AND device and initialised on HOST.
- double GENN_PREFERENCES::asGoodAsZero = 1e-19

  Global variable that is used when detecting close to zero values, for example when setting sparse connectivity from a dense matrix.
- int GENN_PREFERENCES::defaultDevice = 0
- unsigned int GENN_PREFERENCES::neuronBlockSize = 32

  default GPU device; used to determine which GPU to use if chooseDevice is 0 (off)
- unsigned int GENN_PREFERENCES::synapseBlockSize = 32
- unsigned int GENN_PREFERENCES::learningBlockSize = 32
- unsigned int GENN_PREFERENCES::synapseDynamicsBlockSize = 32
- unsigned int GENN_PREFERENCES::initBlockSize = 32
- unsigned int GENN_PREFERENCES::initSparseBlockSize = 32
- unsigned int GENN_PREFERENCES::autoRefractory = 1

  Flag for signalling whether spikes are only reported if thresholdCondition changes from false to true (autoRefractory == 1) or spikes are emitted whenever thresholdCondition is true no matter what.%.
- std::string GENN_PREFERENCES::userCxxFlagsWIN = ""

  Allows users to set specific C++ compiler options they may want to use for all host side code (used for windows platforms)
- std::string GENN_PREFERENCES::userCxxFlagsGNU = ""

  Allows users to set specific C++ compiler options they may want to use for all host side code (used for unix based platforms)
- std::string GENN_PREFERENCES::userNvccFlags = ""

  Allows users to set specific nvcc compiler options they may want to use for all GPU code (identical for windows and unix platforms)
- unsigned int neuronBlkSz

  Global variable containing the GPU block size for the neuron kernel.
- unsigned int synapseBlkSz

  Global variable containing the GPU block size for the synapse kernel.
- unsigned int learnBlkSz

  Global variable containing the GPU block size for the learn kernel.
- unsigned int synDynBlkSz

  Global variable containing the GPU block size for the synapse dynamics kernel.
- unsigned int initBlkSz

  Global variable containing the GPU block size for the initialization kernel.
- unsigned int initSparseBlkSz

  Global variable containing the GPU block size for the sparse initialization kernel.
- cudaDeviceProp ∗ deviceProp
- int theDevice

  Global variable containing the currently selected CUDA device's number.

- int deviceCount

  Global variable containing the number of CUDA devices on this host.
- int hostCount

  Global variable containing the number of hosts within the local compute cluster.

19.32.1 Detailed Description

Global header file containing a few global variables. Part of the code generation section.

19.32.2 Variable Documentation

19.32.2.1 int deviceCount

Global variable containing the number of CUDA devices on this host.

19.32.2.2 cudaDeviceProp∗ deviceProp

19.32.2.3 int hostCount

Global variable containing the number of hosts within the local compute cluster.

19.32.2.4 unsigned int initBlkSz

Global variable containing the GPU block size for the initialization kernel.

19.32.2.5 unsigned int initSparseBlkSz

Global variable containing the GPU block size for the sparse initialization kernel.

19.32.2.6 unsigned int learnBlkSz

Global variable containing the GPU block size for the learn kernel.

19.32.2.7 unsigned int neuronBlkSz

Global variable containing the GPU block size for the neuron kernel.

19.32.2.8 unsigned int synapseBlkSz

Global variable containing the GPU block size for the synapse kernel.

19.32.2.9 unsigned int synDynBlkSz

Global variable containing the GPU block size for the synapse dynamics kernel.

19.32.2.10 int theDevice

Global variable containing the currently selected CUDA device's number.

19.33 hr_time.cc File Reference

This file contains the implementation of the CStopWatch class that provides a simple timing tool based on the system clock.

```
#include <cstdio>
#include "hr_time.h"
```

Macros

- #define HR_TIMER

19.33.1 Detailed Description

This file contains the implementation of the CStopWatch class that provides a simple timing tool based on the system clock.

19.33.2 Macro Definition Documentation

19.33.2.1 #define HR_TIMER

19.34 hr_time.h File Reference

This header file contains the definition of the CStopWatch class that implements a simple timing tool using the system clock.

```
#include <sys/time.h>
```

Classes

- struct stopWatch
- class CStopWatch

    Helper class for timing sections of host code in a cross-plarform manner.

19.34.1 Detailed Description

This header file contains the definition of the CStopWatch class that implements a simple timing tool using the system clock.

19.35 initVarSnippet.cc File Reference

```
#include "initVarSnippet.h"
```

Functions

- IMPLEMENT_SNIPPET (InitVarSnippet::Uninitialised)
- IMPLEMENT_SNIPPET (InitVarSnippet::Constant)
- IMPLEMENT_SNIPPET (InitVarSnippet::Uniform)
- IMPLEMENT_SNIPPET (InitVarSnippet::Normal)
- IMPLEMENT_SNIPPET (InitVarSnippet::Exponential)

19.35.1 Function Documentation

19.35.1.1 IMPLEMENT_SNIPPET (InitVarSnippet::Uninitialised)

19.35.1.2 IMPLEMENT_SNIPPET (InitVarSnippet::Constant)

19.35.1.3 IMPLEMENT_SNIPPET (InitVarSnippet::Uniform)

19.35.1.4 IMPLEMENT_SNIPPET (InitVarSnippet::Normal)

19.35.1.5 IMPLEMENT_SNIPPET (InitVarSnippet::Exponential)

19.36 initVarSnippet.h File Reference

```
#include "snippet.h"
```

Classes

- class InitVarSnippet::Base
- class InitVarSnippet::Uninitialised

    Used to mark variables as uninitialised - no initialisation code will be run.
- class InitVarSnippet::Constant

    Initialises variable to a constant value.
- class InitVarSnippet::Uniform

    Initialises variable by sampling from the uniform distribution.
- class InitVarSnippet::Normal

    Initialises variable by sampling from the normal distribution.
- class InitVarSnippet::Exponential

    Initialises variable by sampling from the exponential distribution.

Namespaces

- InitVarSnippet

    Base class for all value initialisation snippets.

Macros

- #define SET_CODE(CODE) virtual std::string getCode() const{ return CODE; }

19.36.1 Macro Definition Documentation

19.36.1.1 #define SET_CODE(*CODE*) virtual std::string getCode() const{ return CODE; }

19.37 modelSpec.cc File Reference

19.38 modelSpec.cc File Reference

```
#include "codeGenUtils.h"
#include "global.h"
#include "modelSpec.h"
#include "standardSubstitutions.h"
#include "utils.h"
#include <cstdio>
#include <cmath>
#include <cassert>
#include <algorithm>
```

Macros

- #define MODELSPEC_CC

Functions

- void initGeNN ()

  Method for GeNN initialisation (by preparing standard models)

Variables

- unsigned int GeNNReady = 0

19.38.1 Macro Definition Documentation

19.38.1.1 #define MODELSPEC_CC

19.38.2 Function Documentation

19.38.2.1 void initGeNN ( )

Method for GeNN initialisation (by preparing standard models)

19.38.3 Variable Documentation

19.38.3.1 unsigned int GeNNReady = 0

19.39 modelSpec.h File Reference

Header file that contains the class (struct) definition of neuronModel for defining a neuron model and the class definition of NNmodel for defining a neuronal network model. Part of the code generation and generated code sections.

```
#include "neuronGroup.h"
#include "synapseGroup.h"
#include "utils.h"
#include <map>
#include <set>
#include <string>
#include <vector>
```

Classes

- class NNmodel

Macros

- #define _MODELSPEC_H_

  macro for avoiding multiple inclusion during compilation
- #define NO_DELAY 0

  Macro used to indicate no synapse delay for the group (only one queue slot will be generated)
- #define NOLEARNING 0

  Macro attaching the label "NOLEARNING" to flag 0.
- #define LEARNING 1

  Macro attaching the label "LEARNING" to flag 1.
- #define EXITSYN 0

  Macro attaching the label "EXITSYN" to flag 0 (excitatory synapse)

- #define INHIBSYN 1

  Macro attaching the label "INHIBSYN" to flag 1 (inhibitory synapse)
- #define CPU 0

  Macro attaching the label "CPU" to flag 0.
- #define GPU 1

  Macro attaching the label "GPU" to flag 1.
- #define AUTODEVICE -1

  Macro attaching the label AUTODEVICE to flag -1. Used by setGPUDevice.

Enumerations

- enum SynapseConnType { ALLTOALL, DENSE, SPARSE }
- enum SynapseGType { INDIVIDUALG, GLOBALG, INDIVIDUALID }
- enum FloatType { , GENN_LONG_DOUBLE }

Functions

- void initGeNN ()

  Method for GeNN initialisation (by preparing standard models)
- template<typename Snippet >
  NewModels::VarInit initVar (const typename Snippet::ParamValues &params)
- NewModels::VarInit uninitialisedVar ()

Variables

- unsigned int GeNNReady

19.39.1 Detailed Description

Header file that contains the class (struct) definition of neuronModel for defining a neuron model and the class definition of NNmodel for defining a neuronal network model. Part of the code generation and generated code sections.

19.39.2 Macro Definition Documentation

19.39.2.1 #define _MODELSPEC_H_

macro for avoiding multiple inclusion during compilation

19.39.2.2 #define AUTODEVICE -1

Macro attaching the label AUTODEVICE to flag -1. Used by setGPUDevice.

19.39.2.3 #define CPU 0

Macro attaching the label "CPU" to flag 0.

19.39.2.4 #define EXITSYN 0

Macro attaching the label "EXITSYN" to flag 0 (excitatory synapse)

19.39.2.5 #define GPU 1

Macro attaching the label "GPU" to flag 1.

19.39.2.6 #define INHIBSYN 1

Macro attaching the label "INHIBSYN" to flag 1 (inhibitory synapse)

19.39.2.7 #define LEARNING 1

Macro attaching the label "LEARNING" to flag 1.

19.39.2.8 #define NO_DELAY 0

Macro used to indicate no synapse delay for the group (only one queue slot will be generated)

19.39.2.9 #define NOLEARNING 0

Macro attaching the label "NOLEARNING" to flag 0.

19.39.3 Enumeration Type Documentation

19.39.3.1 enum **FloatType**

Enumerator

GENN_LONG_DOUBLE

19.39.3.2 enum **SynapseConnType**

Enumerator

ALLTOALL

DENSE

SPARSE

19.39.3.3 enum **SynapseGType**

Enumerator

INDIVIDUALG

GLOBALG

INDIVIDUALID

19.39.4 Function Documentation

19.39.4.1 void initGeNN ( )

Method for GeNN initialisation (by preparing standard models)

19.39.4.2 template<typename Snippet > **NewModels::VarInit** initVar (const typename Snippet::ParamValues & *params*) [inline]

19.39.4.3 **NewModels::VarInit** uninitialisedVar ( ) [inline]

19.39.5 Variable Documentation

19.39.5.1 unsigned int GeNNReady

19.40 neuronGroup.cc File Reference

```
#include "neuronGroup.h"
#include <algorithm>
#include <cmath>
#include "codeGenUtils.h"
#include "standardSubstitutions.h"
#include "synapseGroup.h"
#include "utils.h"
```

19.41 neuronGroup.h File Reference

```
#include <map>
#include <set>
#include <string>
#include <vector>
#include "global.h"
#include "newNeuronModels.h"
#include "variableMode.h"
```

Classes

- class NeuronGroup

19.42 neuronModels.cc File Reference

```
#include "codeGenUtils.h"
#include "neuronModels.h"
#include "extra_neurons.h"
```

Macros

- #define NEURONMODELS_CC

Functions

- void prepareStandardModels ()

  Function that defines standard neuron models.

Variables

- vector< neuronModel > nModels

  Global C++ vector containing all neuron model descriptions.
- unsigned int MAPNEURON

  variable attaching the name "MAPNEURON"
- unsigned int POISSONNEURON

  variable attaching the name "POISSONNEURON"
- unsigned int TRAUBMILES_FAST

  variable attaching the name "TRAUBMILES_FAST"

- unsigned int TRAUBMILES_ALTERNATIVE

  variable attaching the name "TRAUBMILES_ALTERNATIVE"
- unsigned int TRAUBMILES_SAFE

  variable attaching the name "TRAUBMILES_SAFE"
- unsigned int TRAUBMILES

  variable attaching the name "TRAUBMILES"
- unsigned int TRAUBMILES_PSTEP

  variable attaching the name "TRAUBMILES_PSTEP"
- unsigned int IZHIKEVICH

  variable attaching the name "IZHIKEVICH"
- unsigned int IZHIKEVICH_V

  variable attaching the name "IZHIKEVICH_V"
- unsigned int SPIKESOURCE

  variable attaching the name "SPIKESOURCE"

19.42.1 Macro Definition Documentation

19.42.1.1 #define NEURONMODELS_CC

19.42.2 Function Documentation

19.42.2.1 void prepareStandardModels ( )

Function that defines standard neuron models.

The neuron models are defined and added to the C++ vector nModels that is holding all neuron model descriptions. User defined neuron models can be appended to this vector later in (a) separate function(s).

19.42.3 Variable Documentation

19.42.3.1 unsigned int IZHIKEVICH

variable attaching the name "IZHIKEVICH"

19.42.3.2 unsigned int IZHIKEVICH_V

variable attaching the name "IZHIKEVICH_V"

19.42.3.3 unsigned int MAPNEURON

variable attaching the name "MAPNEURON"

19.42.3.4 vector<neuronModel> nModels

Global C++ vector containing all neuron model descriptions.

19.42.3.5 unsigned int POISSONNEURON

variable attaching the name "POISSONNEURON"

19.42.3.6 unsigned int SPIKESOURCE

variable attaching the name "SPIKESOURCE"

19.42.3.7 unsigned int TRAUBMILES

variable attaching the name "TRAUBMILES"

19.42.3.8 unsigned int TRAUBMILES_ALTERNATIVE

variable attaching the name "TRAUBMILES_ALTERNATIVE"

19.42.3.9 unsigned int TRAUBMILES_FAST

variable attaching the name "TRAUBMILES_FAST"

19.42.3.10 unsigned int TRAUBMILES_PSTEP

variable attaching the name "TRAUBMILES_PSTEP"

19.42.3.11 unsigned int TRAUBMILES_SAFE

variable attaching the name "TRAUBMILES_SAFE"

19.43 neuronModels.h File Reference

```
#include "dpclass.h"
#include <string>
#include <vector>
```

Classes

- class neuronModel

  class for specifying a neuron model.
- class rulkovdp

  Class defining the dependent parameters of the Rulkov map neuron.

Functions

- void prepareStandardModels ()

  Function that defines standard neuron models.

Variables

- vector< neuronModel > nModels

  Global C++ vector containing all neuron model descriptions.
- unsigned int MAPNEURON

  variable attaching the name "MAPNEURON"
- unsigned int POISSONNEURON

  variable attaching the name "POISSONNEURON"
- unsigned int TRAUBMILES_FAST

  variable attaching the name "TRAUBMILES_FAST"
- unsigned int TRAUBMILES_ALTERNATIVE

  variable attaching the name "TRAUBMILES_ALTERNATIVE"
- unsigned int TRAUBMILES_SAFE

  variable attaching the name "TRAUBMILES_SAFE"
- unsigned int TRAUBMILES

  variable attaching the name "TRAUBMILES"
- unsigned int TRAUBMILES_PSTEP

  variable attaching the name "TRAUBMILES_PSTEP"

- unsigned int IZHIKEVICH

  variable attaching the name "IZHIKEVICH"
- unsigned int IZHIKEVICH_V

  variable attaching the name "IZHIKEVICH_V"
- unsigned int SPIKESOURCE

  variable attaching the name "SPIKESOURCE"
- const unsigned int MAXNRN = 7

19.43.1 Function Documentation

19.43.1.1 void prepareStandardModels ( )

Function that defines standard neuron models.

The neuron models are defined and added to the C++ vector nModels that is holding all neuron model descriptions. User defined neuron models can be appended to this vector later in (a) separate function(s).

19.43.2 Variable Documentation

19.43.2.1 unsigned int IZHIKEVICH

variable attaching the name "IZHIKEVICH"

19.43.2.2 unsigned int IZHIKEVICH_V

variable attaching the name "IZHIKEVICH_V"

19.43.2.3 unsigned int MAPNEURON

variable attaching the name "MAPNEURON"

19.43.2.4 const unsigned int MAXNRN = 7

19.43.2.5 vector<neuronModel> nModels

Global C++ vector containing all neuron model descriptions.

19.43.2.6 unsigned int POISSONNEURON

variable attaching the name "POISSONNEURON"

19.43.2.7 unsigned int SPIKESOURCE

variable attaching the name "SPIKESOURCE"

19.43.2.8 unsigned int TRAUBMILES

variable attaching the name "TRAUBMILES"

19.43.2.9 unsigned int TRAUBMILES_ALTERNATIVE

variable attaching the name "TRAUBMILES_ALTERNATIVE"

19.43.2.10 unsigned int TRAUBMILES_FAST

variable attaching the name "TRAUBMILES_FAST"

19.43.2.11 unsigned int TRAUBMILES_PSTEP

variable attaching the name "TRAUBMILES_PSTEP"

19.43.2.12 unsigned int TRAUBMILES_SAFE

variable attaching the name "TRAUBMILES_SAFE"

19.44 newModels.h File Reference

```
#include <algorithm>
#include <string>
#include <vector>
#include <cassert>
#include "snippet.h"
#include "initVarSnippet.h"
```

Classes

- class NewModels::VarInit
- class NewModels::VarInitContainerBase< NumVars >
- class NewModels::VarInitContainerBase< 0 >
- class NewModels::Base

  Base class for all models.
- class NewModels::LegacyWrapper< ModelBase, LegacyModelType, ModelArray >

  Wrapper around old-style models stored in global arrays and referenced by index.

Namespaces

- NewModels

Macros

- #define DECLARE_MODEL(TYPE, NUM_PARAMS, NUM_VARS)
- #define IMPLEMENT_MODEL(TYPE) IMPLEMENT_SNIPPET(TYPE)
- #define SET_VARS(...) virtual StringPairVec getVars() const{ return __VA_ARGS__; }

19.44.1 Macro Definition Documentation

19.44.1.1 #define DECLARE_MODEL(*TYPE*, *NUM_PARAMS*, *NUM_VARS*)

Value:

```
DECLARE_SNIPPET(TYPE, NUM_PARAMS)                                   \
    typedef NewModels::VarInitContainerBase<NUM_VARS> VarValues;
            \
```

19.44.1.2 #define IMPLEMENT_MODEL(*TYPE*) IMPLEMENT_SNIPPET(TYPE)

19.44.1.3 #define SET_VARS(*...*) virtual StringPairVec getVars() const{ return __VA_ARGS__; }

19.45 newNeuronModels.cc File Reference

```
#include "newNeuronModels.h"
```

Functions

- IMPLEMENT_MODEL (NeuronModels::RulkovMap)
- IMPLEMENT_MODEL (NeuronModels::Izhikevich)
- IMPLEMENT_MODEL (NeuronModels::IzhikevichVariable)
- IMPLEMENT_MODEL (NeuronModels::SpikeSource)
- IMPLEMENT_MODEL (NeuronModels::Poisson)
- IMPLEMENT_MODEL (NeuronModels::PoissonNew)
- IMPLEMENT_MODEL (NeuronModels::TraubMiles)
- IMPLEMENT_MODEL (NeuronModels::TraubMilesFast)
- IMPLEMENT_MODEL (NeuronModels::TraubMilesAlt)
- IMPLEMENT_MODEL (NeuronModels::TraubMilesNStep)

19.45.1 Function Documentation

19.45.1.1 IMPLEMENT_MODEL (NeuronModels::RulkovMap)

19.45.1.2 IMPLEMENT_MODEL (NeuronModels::Izhikevich)

19.45.1.3 IMPLEMENT_MODEL (NeuronModels::IzhikevichVariable)

19.45.1.4 IMPLEMENT_MODEL (NeuronModels::SpikeSource)

19.45.1.5 IMPLEMENT_MODEL (NeuronModels::Poisson)

19.45.1.6 IMPLEMENT_MODEL (NeuronModels::PoissonNew)

19.45.1.7 IMPLEMENT_MODEL (NeuronModels::TraubMiles)

19.45.1.8 IMPLEMENT_MODEL (NeuronModels::TraubMilesFast)

19.45.1.9 IMPLEMENT_MODEL (NeuronModels::TraubMilesAlt)

19.45.1.10 IMPLEMENT_MODEL (NeuronModels::TraubMilesNStep)

19.46 newNeuronModels.h File Reference

```
#include <array>
#include <functional>
#include <string>
#include <tuple>
#include <vector>
#include "codeGenUtils.h"
#include "neuronModels.h"
#include "newModels.h"
```

Classes

- class NeuronModels::Base

  Base class for all neuron models.
- class NeuronModels::LegacyWrapper

  Wrapper around legacy weight update models stored in nModels array of neuronModel objects.
- class NeuronModels::RulkovMap

  Rulkov Map neuron.
- class NeuronModels::Izhikevich

Izhikevich neuron with fixed parameters [1].
- class NeuronModels::IzhikevichVariable

  Izhikevich neuron with variable parameters [1].
- class NeuronModels::SpikeSource

  Empty neuron which allows setting spikes from external sources.
- class NeuronModels::Poisson

  Poisson neurons.
- class NeuronModels::PoissonNew

  Poisson neurons.
- class NeuronModels::TraubMiles

  Hodgkin-Huxley neurons with Traub & Miles algorithm.
- class NeuronModels::TraubMilesFast

  Hodgkin-Huxley neurons with Traub & Miles algorithm: Original fast implementation, using 25 inner iterations.
- class NeuronModels::TraubMilesAlt

  Hodgkin-Huxley neurons with Traub & Miles algorithm.
- class NeuronModels::TraubMilesNStep

  Hodgkin-Huxley neurons with Traub & Miles algorithm.

Namespaces

- NeuronModels

Macros

- #define SET_SIM_CODE(SIM_CODE) virtual std::string getSimCode() const{ return SIM_CODE; }
- #define SET_THRESHOLD_CONDITION_CODE(THRESHOLD_CONDITION_CODE) virtual std::string getThresholdConditionCode() const{ return THRESHOLD_CONDITION_CODE; }
- #define SET_RESET_CODE(RESET_CODE) virtual std::string getResetCode() const{ return RESET_CO↩DE; }
- #define SET_SUPPORT_CODE(SUPPORT_CODE) virtual std::string getSupportCode() const{ return SU↩PPORT_CODE; }
- #define SET_EXTRA_GLOBAL_PARAMS(...) virtual StringPairVec getExtraGlobalParams() const{ return ↩ __VA_ARGS__; }
- #define SET_ADDITIONAL_INPUT_VARS(...) virtual NameTypeValVec getAdditionalInputVars() const{ return __VA_ARGS__; }

19.46.1 Macro Definition Documentation

19.46.1.1 #define SET_ADDITIONAL_INPUT_VARS(...) virtual NameTypeValVec getAdditionalInputVars() const{ return __VA_ARGS__; }

19.46.1.2 #define SET_EXTRA_GLOBAL_PARAMS(...) virtual StringPairVec getExtraGlobalParams() const{ return __VA_ARGS__; }

19.46.1.3 #define SET_RESET_CODE(*RESET_CODE*) virtual std::string getResetCode() const{ return RESET_CODE; }

19.46.1.4 #define SET_SIM_CODE(*SIM_CODE*) virtual std::string getSimCode() const{ return SIM_CODE; }

19.46.1.5 #define SET_SUPPORT_CODE(*SUPPORT_CODE*) virtual std::string getSupportCode() const{ return SUPPORT_CODE; }

19.46.1.6 #define SET_THRESHOLD_CONDITION_CODE(*THRESHOLD_CONDITION_CODE*) virtual std::string getThresholdConditionCode() const{ return THRESHOLD_CONDITION_CODE; }

19.47 newPostsynapticModels.cc File Reference

```
#include "newPostsynapticModels.h"
```

Functions

- IMPLEMENT_MODEL (PostsynapticModels::ExpCond)
- IMPLEMENT_MODEL (PostsynapticModels::DeltaCurr)

19.47.1 Function Documentation

19.47.1.1 IMPLEMENT_MODEL (PostsynapticModels::ExpCond)

19.47.1.2 IMPLEMENT_MODEL (PostsynapticModels::DeltaCurr)

19.48 newPostsynapticModels.h File Reference

```
#include "newModels.h"
#include "postSynapseModels.h"
```

Classes

- class PostsynapticModels::Base

  Base class for all postsynaptic models.
- class PostsynapticModels::LegacyWrapper
- class PostsynapticModels::ExpCond

  Exponential decay with synaptic input treated as a conductance value.
- class PostsynapticModels::DeltaCurr

  Simple delta current synapse.

Namespaces

- PostsynapticModels

Macros

- #define SET_DECAY_CODE(DECAY_CODE) virtual std::string getDecayCode() const{ return DECAY_C↩ ODE; }
- #define SET_CURRENT_CONVERTER_CODE(CURRENT_CONVERTER_CODE) virtual std::string get↩ ApplyInputCode() const{ return "$(Isyn) += " CURRENT_CONVERTER_CODE ";"; }
- #define SET_APPLY_INPUT_CODE(APPLY_INPUT_CODE) virtual std::string getApplyInputCode() const{ return APPLY_INPUT_CODE; }
- #define SET_SUPPORT_CODE(SUPPORT_CODE) virtual std::string getSupportCode() const{ return SU↩ PPORT_CODE; }

19.48.1 Macro Definition Documentation

19.48.1.1 #define SET_APPLY_INPUT_CODE(*APPLY_INPUT_CODE*) virtual std::string getApplyInputCode() const{ return APPLY_INPUT_CODE; }

19.48.1.2 #define SET_CURRENT_CONVERTER_CODE(*CURRENT_CONVERTER_CODE*) virtual std::string getApplyInputCode() const{ return "$(Isyn) += " CURRENT_CONVERTER_CODE ";"; }

19.48.1.3 #define SET_DECAY_CODE(*DECAY_CODE*) virtual std::string getDecayCode() const{ return DECAY_CODE; }

19.48.1.4 #define SET_SUPPORT_CODE(*SUPPORT_CODE*) virtual std::string getSupportCode() const{ return SUPPORT_CODE; }

19.49 newWeightUpdateModels.cc File Reference

```
#include "newWeightUpdateModels.h"
```

Functions

- IMPLEMENT_MODEL (WeightUpdateModels::StaticPulse)
- IMPLEMENT_MODEL (WeightUpdateModels::StaticGraded)
- IMPLEMENT_MODEL (WeightUpdateModels::PiecewiseSTDP)

19.49.1 Function Documentation

19.49.1.1 IMPLEMENT_MODEL (WeightUpdateModels::StaticPulse)

19.49.1.2 IMPLEMENT_MODEL (WeightUpdateModels::StaticGraded)

19.49.1.3 IMPLEMENT_MODEL (WeightUpdateModels::PiecewiseSTDP)

19.50 newWeightUpdateModels.h File Reference

```
#include "newModels.h"
#include "synapseModels.h"
```

Classes

- class WeightUpdateModels::Base

  Base class for all weight update models.
- class WeightUpdateModels::LegacyWrapper

  Wrapper around legacy weight update models stored in weightUpdateModels array of weightUpdateModel objects.
- class WeightUpdateModels::StaticPulse

  Pulse-coupled, static synapse.
- class WeightUpdateModels::StaticGraded

  Graded-potential, static synapse.
- class WeightUpdateModels::PiecewiseSTDP

  This is a simple STDP rule including a time delay for the finite transmission speed of the synapse.

Namespaces

- WeightUpdateModels

Macros

- #define SET_SIM_CODE(SIM_CODE) virtual std::string getSimCode() const{ return SIM_CODE; }
- #define SET_EVENT_CODE(EVENT_CODE) virtual std::string getEventCode() const{ return EVENT_CODE; }
- #define SET_LEARN_POST_CODE(LEARN_POST_CODE) virtual std::string getLearnPostCode() const{ return LEARN_POST_CODE; }
- #define SET_SYNAPSE_DYNAMICS_CODE(SYNAPSE_DYNAMICS_CODE) virtual std::string getSynapseDynamicsCode() const{ return SYNAPSE_DYNAMICS_CODE; }
- #define SET_EVENT_THRESHOLD_CONDITION_CODE(EVENT_THRESHOLD_CONDITION_CODE) virtual std::string getEventThresholdConditionCode() const{ return EVENT_THRESHOLD_CONDITION_CODE; }
- #define SET_SIM_SUPPORT_CODE(SIM_SUPPORT_CODE) virtual std::string getSimSupportCode() const{ return SIM_SUPPORT_CODE; }
- #define SET_LEARN_POST_SUPPORT_CODE(LEARN_POST_SUPPORT_CODE) virtual std::string getLearnPostSupportCode() const{ return LEARN_POST_SUPPORT_CODE; }
- #define SET_SYNAPSE_DYNAMICS_SUPPORT_CODE(SYNAPSE_DYNAMICS_SUPPORT_CODE) virtual std::string getSynapseDynamicsSuppportCode() const{ return SYNAPSE_DYNAMICS_SUPPORT_CODE; }
- #define SET_EXTRA_GLOBAL_PARAMS(...) virtual StringPairVec getExtraGlobalParams() const{ return __VA_ARGS__; }
- #define SET_NEEDS_PRE_SPIKE_TIME(PRE_SPIKE_TIME_REQUIRED) virtual bool isPreSpikeTimeRequired() const{ return PRE_SPIKE_TIME_REQUIRED; }
- #define SET_NEEDS_POST_SPIKE_TIME(POST_SPIKE_TIME_REQUIRED) virtual bool isPostSpikeTimeRequired() const{ return POST_SPIKE_TIME_REQUIRED; }

19.50.1 Macro Definition Documentation

19.50.1.1 #define SET_EVENT_CODE(*EVENT_CODE*) virtual std::string getEventCode() const{ return EVENT_CODE; }

19.50.1.2 #define SET_EVENT_THRESHOLD_CONDITION_CODE(*EVENT_THRESHOLD_CONDITION_CODE*) virtual std::string getEventThresholdConditionCode() const{ return EVENT_THRESHOLD_CONDITION_CODE; }

19.50.1.3 #define SET_EXTRA_GLOBAL_PARAMS(*...*) virtual StringPairVec getExtraGlobalParams() const{ return __VA_ARGS__; }

19.50.1.4 #define SET_LEARN_POST_CODE(*LEARN_POST_CODE*) virtual std::string getLearnPostCode() const{ return LEARN_POST_CODE; }

19.50.1.5 #define SET_LEARN_POST_SUPPORT_CODE(*LEARN_POST_SUPPORT_CODE*) virtual std::string getLearnPostSupportCode() const{ return LEARN_POST_SUPPORT_CODE; }

19.50.1.6 #define SET_NEEDS_POST_SPIKE_TIME(*POST_SPIKE_TIME_REQUIRED*) virtual bool isPostSpikeTimeRequired() const{ return POST_SPIKE_TIME_REQUIRED; }

19.50.1.7 #define SET_NEEDS_PRE_SPIKE_TIME(*PRE_SPIKE_TIME_REQUIRED*) virtual bool isPreSpikeTimeRequired() const{ return PRE_SPIKE_TIME_REQUIRED; }

19.50.1.8 #define SET_SIM_CODE(*SIM_CODE*) virtual std::string getSimCode() const{ return SIM_CODE; }

19.50.1.9 #define SET_SIM_SUPPORT_CODE(*SIM_SUPPORT_CODE*) virtual std::string getSimSupportCode() const{ return SIM_SUPPORT_CODE; }

19.50.1.10 #define SET_SYNAPSE_DYNAMICS_CODE(*SYNAPSE_DYNAMICS_CODE*) virtual std::string getSynapseDynamicsCode() const{ return SYNAPSE_DYNAMICS_CODE; }

19.50.1.11 #define SET_SYNAPSE_DYNAMICS_SUPPORT_CODE(*SYNAPSE_DYNAMICS_SUPPORT_CODE*) virtual std::string getSynapseDynamicsSuppportCode() const{ return SYNAPSE_DYNAMICS_SUPPORT_CODE; }

19.51 postSynapseModels.cc File Reference

```
#include "codeGenUtils.h"
#include "postSynapseModels.h"
#include "extra_postsynapses.h"
```

Macros

- #define POSTSYNAPSEMODELS_CC

Functions

- void preparePostSynModels ()

    Function that prepares the standard post-synaptic models, including their variables, parameters, dependent parameters and code strings.

Variables

- vector< postSynModel > postSynModels

    Global C++ vector containing all post-synaptic update model descriptions.
- unsigned int EXPDECAY
- unsigned int IZHIKEVICH_PS

19.51.1 Macro Definition Documentation

19.51.1.1 #define POSTSYNAPSEMODELS_CC

19.51.2 Function Documentation

19.51.2.1 void preparePostSynModels ( )

Function that prepares the standard post-synaptic models, including their variables, parameters, dependent parameters and code strings.

19.51.3 Variable Documentation

19.51.3.1 unsigned int EXPDECAY

19.51.3.2 unsigned int IZHIKEVICH_PS

19.51.3.3 vector<postSynModel> postSynModels

Global C++ vector containing all post-synaptic update model descriptions.

19.52 postSynapseModels.h File Reference

```
#include "dpclass.h"
#include <string>
#include <vector>
#include <cmath>
```

Classes

- class postSynModel

  Class to hold the information that defines a post-synaptic model (a model of how synapses affect post-synaptic neuron variables, classically in the form of a synaptic current). It also allows to define an equation for the dynamics that can be applied to the summed synaptic input variable "insyn".
- class expDecayDp

  Class defining the dependent parameter for exponential decay.

Functions

- void preparePostSynModels ()

  Function that prepares the standard post-synaptic models, including their variables, parameters, dependent parameters and code strings.

Variables

- vector< postSynModel > postSynModels

  Global C++ vector containing all post-synaptic update model descriptions.
- unsigned int EXPDECAY
- unsigned int IZHIKEVICH_PS
- const unsigned int MAXPOSTSYN = 2

19.52.1 Function Documentation

19.52.1.1 void preparePostSynModels ( )

Function that prepares the standard post-synaptic models, including their variables, parameters, dependent parameters and code strings.

19.52.2 Variable Documentation

19.52.2.1 unsigned int EXPDECAY

19.52.2.2 unsigned int IZHIKEVICH_PS

19.52.2.3 const unsigned int MAXPOSTSYN = 2

19.52.2.4 vector<postSynModel> postSynModels

Global C++ vector containing all post-synaptic update model descriptions.

19.53 snippet.h File Reference

```
#include <functional>
#include <string>
#include <vector>
```

Classes

- class Snippet::ValueBase< NumVars >
- class Snippet::ValueBase< 0 >

- class Snippet::Base

  Base class for all code snippets.

Namespaces

- Snippet

Macros

- #define DECLARE_SNIPPET(TYPE, NUM_PARAMS)
- #define IMPLEMENT_SNIPPET(TYPE) TYPE ∗TYPE::s_Instance = NULL
- #define SET_PARAM_NAMES(...) virtual StringVec getParamNames() const{ return __VA_ARGS__; }
- #define SET_DERIVED_PARAMS(...) virtual DerivedParamVec getDerivedParams() const{ return __VA_A↩ RGS__; }

19.53.1 Macro Definition Documentation

19.53.1.1 #define DECLARE_SNIPPET(*TYPE, NUM_PARAMS*)

Value:

```
private:                                                \
    static TYPE *s_Instance;                            \
public:                                                 \
    static const TYPE *getInstance()                    \
    {                                                   \
        if(s_Instance == NULL)                          \
        {                                               \
            s_Instance = new TYPE;                      \
        }                                               \
        return s_Instance;                              \
    }                                                   \
    typedef Snippet::ValueBase<NUM_PARAMS> ParamValues; \
```

19.53.1.2 #define IMPLEMENT_SNIPPET(*TYPE*) TYPE ∗TYPE::s_Instance = NULL

19.53.1.3 #define SET_DERIVED_PARAMS(*...*) virtual DerivedParamVec getDerivedParams() const{ return __VA_ARGS__; }

19.53.1.4 #define SET_PARAM_NAMES(*...*) virtual StringVec getParamNames() const{ return __VA_ARGS__; }

19.54 sparseProjection.h File Reference

Classes

- struct SparseProjection

  class (struct) for defining a spars connectivity projection

19.55 sparseUtils.cc File Reference

```
#include "sparseUtils.h"
#include "utils.h"
#include <vector>
```

Macros

- #define SPARSEUTILS_CC

Functions

- void createPosttoPreArray (unsigned int preN, unsigned int postN, SparseProjection ∗C)

    Utility to generate the SPARSE array structure with post-to-pre arrangement from the original pre-to-post arrangement where postsynaptic feedback is necessary (learning etc)

- void createPreIndices (unsigned int preN, unsigned int postN, SparseProjection ∗C)

    Function to create the mapping from the normal index array "ind" to the "reverse" array revInd, i.e. the inverse mapping of remap. This is needed if SynapseDynamics accesses pre-synaptic variables.

- void initializeSparseArray (SparseProjection C, unsigned int ∗dInd, unsigned int ∗dIndInG, unsigned int preN)

    Function for initializing conductance array indices for sparse matrices on the GPU (by copying the values from the host)

- void initializeSparseArrayRev (SparseProjection C, unsigned int ∗dRevInd, unsigned int ∗dRevIndInG, unsigned int ∗dRemap, unsigned int postN)

    Function for initializing reversed conductance array indices for sparse matrices on the GPU (by copying the values from the host)

- void initializeSparseArrayPreInd (SparseProjection C, unsigned int ∗dPreInd)

    Function for initializing reversed conductance arrays presynaptic indices for sparse matrices on the GPU (by copying the values from the host)

19.55.1 Macro Definition Documentation

19.55.1.1 #define SPARSEUTILS_CC

19.55.2 Function Documentation

19.55.2.1 void createPosttoPreArray (unsigned int *preN,* unsigned int *postN,* SparseProjection ∗ *C*)

Utility to generate the SPARSE array structure with post-to-pre arrangement from the original pre-to-post arrangement where postsynaptic feedback is necessary (learning etc)

19.55.2.2 void createPreIndices (unsigned int *preN,* unsigned int *postN,* SparseProjection ∗ *C*)

Function to create the mapping from the normal index array "ind" to the "reverse" array revInd, i.e. the inverse mapping of remap. This is needed if SynapseDynamics accesses pre-synaptic variables.

19.55.2.3 void initializeSparseArray (SparseProjection *C,* unsigned int ∗ *dInd,* unsigned int ∗ *dIndInG,* unsigned int *preN*)

Function for initializing conductance array indices for sparse matrices on the GPU (by copying the values from the host)

19.55.2.4 void initializeSparseArrayPreInd (SparseProjection *C,* unsigned int ∗ *dPreInd*)

Function for initializing reversed conductance arrays presynaptic indices for sparse matrices on the GPU (by copying the values from the host)

19.55.2.5 void initializeSparseArrayRev (SparseProjection *C,* unsigned int ∗ *dRevInd,* unsigned int ∗ *dRevIndInG,* unsigned int ∗ *dRemap,* unsigned int *postN*)

Function for initializing reversed conductance array indices for sparse matrices on the GPU (by copying the values from the host)

19.56 sparseUtils.h File Reference

```
#include "sparseProjection.h"
#include "global.h"
#include <cstdlib>
#include <cstdio>
#include <string>
#include <cmath>
```

Functions

- template<class DATATYPE >
  unsigned int countEntriesAbove (DATATYPE ∗Array, int sz, double includeAbove)

  Utility to count how many entries above a specified value exist in a float array.
- template<class DATATYPE >
  DATATYPE getG (DATATYPE ∗wuvar, SparseProjection ∗sparseStruct, int x, int y)

  DEPRECATED Utility to get a synapse weight from a SPARSE structure by x,y coordinates NB: as the SparseProjection struct doesnt hold the preN size (it should!) it is not possible to check the parameter validity. This fn may therefore crash unless user knows max poss X.
- template<class DATATYPE >
  float getSparseVar (DATATYPE ∗wuvar, SparseProjection ∗sparseStruct, unsigned int x, unsigned int y)
- template<class DATATYPE >
  void setSparseConnectivityFromDense (DATATYPE ∗wuvar, int preN, int postN, DATATYPE ∗tmp_gRNPN, SparseProjection ∗sparseStruct)

  Function for setting the values of SPARSE connectivity matrix.
- template<class DATATYPE >
  void createSparseConnectivityFromDense (DATATYPE ∗wuvar, int preN, int postN, DATATYPE ∗tmp_gRNPN, SparseProjection ∗sparseStruct, bool runTest)

  Utility to generate the SPARSE connectivity structure from a simple all-to-all array.
- void createPosttoPreArray (unsigned int preN, unsigned int postN, SparseProjection ∗C)

  Utility to generate the SPARSE array structure with post-to-pre arrangement from the original pre-to-post arrangement where postsynaptic feedback is necessary (learning etc)
- void createPreIndices (unsigned int preN, unsigned int postN, SparseProjection ∗C)

  Function to create the mapping from the normal index array "ind" to the "reverse" array revInd, i.e. the inverse mapping of remap. This is needed if SynapseDynamics accesses pre-synaptic variables.
- void initializeSparseArray (SparseProjection C, unsigned int ∗dInd, unsigned int ∗dIndInG, unsigned int preN)

  Function for initializing conductance array indices for sparse matrices on the GPU (by copying the values from the host)
- void initializeSparseArrayRev (SparseProjection C, unsigned int ∗dRevInd, unsigned int ∗dRevIndInG, unsigned int ∗dRemap, unsigned int postN)

  Function for initializing reversed conductance array indices for sparse matrices on the GPU (by copying the values from the host)
- void initializeSparseArrayPreInd (SparseProjection C, unsigned int ∗dPreInd)

  Function for initializing reversed conductance arrays presynaptic indices for sparse matrices on the GPU (by copying the values from the host)

19.56.1 Function Documentation

19.56.1.1 template<class DATATYPE > unsigned int countEntriesAbove (DATATYPE ∗ *Array,* int *sz,* double *includeAbove*)

Utility to count how many entries above a specified value exist in a float array.

19.56.1.2 void createPosttoPreArray (unsigned int *preN*, unsigned int *postN*, SparseProjection ∗ *C*)

Utility to generate the SPARSE array structure with post-to-pre arrangement from the original pre-to-post arrangement where postsynaptic feedback is necessary (learning etc)

19.56.1.3 void createPreIndices (unsigned int *preN*, unsigned int *postN*, SparseProjection ∗ *C*)

Function to create the mapping from the normal index array "ind" to the "reverse" array revInd, i.e. the inverse mapping of remap. This is needed if SynapseDynamics accesses pre-synaptic variables.

19.56.1.4 template<class DATATYPE > void createSparseConnectivityFromDense (DATATYPE ∗ *wuvar*, int *preN*, int *postN*, DATATYPE ∗ *tmp_gRNPN*, SparseProjection ∗ *sparseStruct*, bool *runTest*)

Utility to generate the SPARSE connectivity structure from a simple all-to-all array.

19.56.1.5 template<class DATATYPE > DATATYPE getG (DATATYPE ∗ *wuvar*, SparseProjection ∗ *sparseStruct*, int *x*, int *y*)

DEPRECATED Utility to get a synapse weight from a SPARSE structure by x,y coordinates NB: as the SparseProjection struct doesnt hold the preN size (it should!) it is not possible to check the parameter validity. This fn may therefore crash unless user knows max poss X.

19.56.1.6 template<class DATATYPE > float getSparseVar (DATATYPE ∗ *wuvar*, SparseProjection ∗ *sparseStruct*, unsigned int *x*, unsigned int *y*)

19.56.1.7 void initializeSparseArray (SparseProjection *C*, unsigned int ∗ *dInd*, unsigned int ∗ *dIndInG*, unsigned int *preN*)

Function for initializing conductance array indices for sparse matrices on the GPU (by copying the values from the host)

19.56.1.8 void initializeSparseArrayPreInd (SparseProjection *C*, unsigned int ∗ *dPreInd*)

Function for initializing reversed conductance arrays presynaptic indices for sparse matrices on the GPU (by copying the values from the host)

19.56.1.9 void initializeSparseArrayRev (SparseProjection *C*, unsigned int ∗ *dRevInd*, unsigned int ∗ *dRevIndInG*, unsigned int ∗ *dRemap*, unsigned int *postN*)

Function for initializing reversed conductance array indices for sparse matrices on the GPU (by copying the values from the host)

19.56.1.10 template<class DATATYPE > void setSparseConnectivityFromDense (DATATYPE ∗ *wuvar*, int *preN*, int *postN*, DATATYPE ∗ *tmp_gRNPN*, SparseProjection ∗ *sparseStruct*)

Function for setting the values of SPARSE connectivity matrix.

19.57 standardGeneratedSections.cc File Reference

```
#include "standardGeneratedSections.h"
#include "codeStream.h"
#include "modelSpec.h"
```

19.58 standardGeneratedSections.h File Reference

```
#include <string>
#include "codeGenUtils.h"
#include "newNeuronModels.h"
#include "standardSubstitutions.h"
```

Namespaces

- StandardGeneratedSections

Functions

- void StandardGeneratedSections::neuronOutputInit (CodeStream &os, const NeuronGroup &ng, const std::string &devPrefix)
- void StandardGeneratedSections::neuronLocalVarInit (CodeStream &os, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const std::string &devPrefix, const std::string &localID)
- void StandardGeneratedSections::neuronLocalVarWrite (CodeStream &os, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const std::string &devPrefix, const std::string &localID)
- void StandardGeneratedSections::neuronSpikeEventTest (CodeStream &os, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::string &localID, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

19.59 standardSubstitutions.cc File Reference

```
#include "standardSubstitutions.h"
#include "codeStream.h"
#include "modelSpec.h"
```

19.60 standardSubstitutions.h File Reference

```
#include <string>
#include "codeGenUtils.h"
#include "newNeuronModels.h"
```

Classes

- struct NameIterCtx< Container >

Namespaces

- StandardSubstitutions

Typedefs

- typedef NameIterCtx< NewModels::Base::StringPairVec > VarNameIterCtx
- typedef NameIterCtx< NewModels::Base::DerivedParamVec > DerivedParamNameIterCtx
- typedef NameIterCtx< NewModels::Base::StringPairVec > ExtraGlobalParamNameIterCtx

Functions

- void StandardSubstitutions::postSynapseApplyInput (std::string &psCode, const SynapseGroup *sg, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const DerivedParamNameIterCtx &nmDerivedParams, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

Applies standard set of variable substitutions to postsynaptic model's "apply input" code.

- void StandardSubstitutions::postSynapseDecay (std::string &pdCode, const SynapseGroup ∗sg, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const DerivedParamNameIterCtx &nmDerivedParams, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

  Name of the RNG to use for any probabilistic operations.
- void StandardSubstitutions::neuronThresholdCondition (std::string &thCode, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const DerivedParamNameIterCtx &nmDerivedParams, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

  Applies standard set of variable substitutions to neuron model's "threshold condition" code.
- void StandardSubstitutions::neuronSim (std::string &sCode, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const DerivedParamNameIterCtx &nmDerivedParams, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)
- void StandardSubstitutions::neuronSpikeEventCondition (std::string &eCode, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)
- void StandardSubstitutions::neuronReset (std::string &rCode, const NeuronGroup &ng, const VarNameIterCtx &nmVars, const DerivedParamNameIterCtx &nmDerivedParams, const ExtraGlobalParamNameIterCtx &nmExtraGlobalParams, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)
- void StandardSubstitutions::weightUpdateThresholdCondition (std::string &eCode, const SynapseGroup &sg, const DerivedParamNameIterCtx &wuDerivedParams, const ExtraGlobalParamNameIterCtx &wuExtraGlobalParams, const string &preIdx, const string &postIdx, const string &devPrefix, const std::vector< FunctionTemplate > functions, const std::string &ftype)
- void StandardSubstitutions::weightUpdateSim (std::string &wCode, const SynapseGroup &sg, const VarNameIterCtx &wuVars, const DerivedParamNameIterCtx &wuDerivedParams, const ExtraGlobalParamNameIterCtx &wuExtraGlobalParams, const string &preIdx, const string &postIdx, const string &devPrefix, const std::vector< FunctionTemplate > functions, const std::string &ftype)
- void StandardSubstitutions::weightUpdateDynamics (std::string &SDcode, const SynapseGroup ∗sg, const VarNameIterCtx &wuVars, const DerivedParamNameIterCtx &wuDerivedParams, const ExtraGlobalParamNameIterCtx &wuExtraGlobalParams, const string &preIdx, const string &postIdx, const string &devPrefix, const std::vector< FunctionTemplate > functions, const std::string &ftype)
- void StandardSubstitutions::weightUpdatePostLearn (std::string &code, const SynapseGroup ∗sg, const DerivedParamNameIterCtx &wuDerivedParams, const ExtraGlobalParamNameIterCtx &wuExtraGlobalParams, const string &preIdx, const string &postIdx, const string &devPrefix, const std::vector< FunctionTemplate > functions, const std::string &ftype)
- std::string StandardSubstitutions::initVariable (const NewModels::VarInit &varInit, const std::string &varName, const std::vector< FunctionTemplate > functions, const std::string &ftype, const std::string &rng)

19.60.1 Typedef Documentation

19.60.1.1 typedef NameIterCtx<NewModels::Base::DerivedParamVec> DerivedParamNameIterCtx

19.60.1.2 typedef NameIterCtx<NewModels::Base::StringPairVec> ExtraGlobalParamNameIterCtx

19.60.1.3 typedef NameIterCtx<NewModels::Base::StringPairVec> VarNameIterCtx

19.61 stringUtils.h File Reference

```
#include <string>
#include <sstream>
```

Macros

- #define tS(X) toString(X)

  Macro providing the abbreviated syntax tS() instead of toString().

Functions

- template<class T >
  std::string toString (T t)

  template functions for conversion of various types to C++ strings

19.61.1 Macro Definition Documentation

19.61.1.1 #define tS(X) toString(X)

Macro providing the abbreviated syntax tS() instead of toString().

19.61.2 Function Documentation

19.61.2.1 template<class T > std::string toString (T *t*)

template functions for conversion of various types to C++ strings

19.62 synapseGroup.cc File Reference

```
#include "synapseGroup.h"
#include <algorithm>
#include <cmath>
#include "codeGenUtils.h"
#include "global.h"
#include "standardSubstitutions.h"
#include "utils.h"
```

19.63 synapseGroup.h File Reference

```
#include <map>
#include <set>
#include <string>
#include <vector>
#include "neuronGroup.h"
#include "newPostsynapticModels.h"
#include "newWeightUpdateModels.h"
#include "synapseMatrixType.h"
```

Classes

- class SynapseGroup

19.64 synapseMatrixType.h File Reference

Enumerations

- enum SynapseMatrixConnectivity : unsigned int { SynapseMatrixConnectivity::SPARSE = (1 << 0), SynapseMatrixConnectivity::DENSE = (1 << 1), SynapseMatrixConnectivity::BITMASK = (1 << 2) }

  < Flags defining differnet types of synaptic matrix connectivity
- enum SynapseMatrixWeight : unsigned int { SynapseMatrixWeight::GLOBAL = (1 << 3), SynapseMatrixWeight::INDIVIDUAL = (1 << 4) }
- enum SynapseMatrixType : unsigned int {
  SynapseMatrixType::SPARSE_GLOBALG = static_cast<unsigned int>(SynapseMatrixConnectivity::SPARSE) | static_cast<unsigned int>(SynapseMatrixWeight::GLOBAL), SynapseMatrixType::SPARSE_INDIVIDUALG = static_cast<unsigned int>(SynapseMatrixConnectivity::SPARSE) | static_cast<unsigned int>(SynapseMatrixWeight::INDIVIDUAL), SynapseMatrixType::DENSE_GLOBALG = static_cast<unsigned int>(SynapseMatrixConnectivity::DENSE) | static_cast<unsigned int>(SynapseMatrixWeight::GLOBAL),
  SynapseMatrixType::DENSE_INDIVIDUALG = static_cast<unsigned int>(SynapseMatrixConnectivity::DENSE) | static_cast<unsigned int>(SynapseMatrixWeight::INDIVIDUAL),
  SynapseMatrixType::BITMASK_GLOBALG = static_cast<unsigned int>(SynapseMatrixConnectivity::BITMASK) | static_cast<unsigned int>(SynapseMatrixWeight::GLOBAL) }

Functions

- bool operator& (SynapseMatrixType type, SynapseMatrixConnectivity connType)
- bool operator& (SynapseMatrixType type, SynapseMatrixWeight weightType)

19.64.1 Enumeration Type Documentation

19.64.1.1 enum **SynapseMatrixConnectivity** : unsigned int `[strong]`

< Flags defining differnet types of synaptic matrix connectivity

Enumerator

SPARSE

DENSE

BITMASK

19.64.1.2 enum **SynapseMatrixType** : unsigned int `[strong]`

Enumerator

SPARSE_GLOBALG

SPARSE_INDIVIDUALG

DENSE_GLOBALG

DENSE_INDIVIDUALG

BITMASK_GLOBALG

19.64.1.3 enum **SynapseMatrixWeight** : unsigned int `[strong]`

Enumerator

GLOBAL

INDIVIDUAL

19.64.2 Function Documentation

19.64.2.1 bool operator& (SynapseMatrixType *type*, SynapseMatrixConnectivity *connType*) [inline]

19.64.2.2 bool operator& (SynapseMatrixType *type*, SynapseMatrixWeight *weightType*) [inline]

19.65 synapseModels.cc File Reference

```
#include "codeGenUtils.h"
#include "synapseModels.h"
#include "extra_weightupdates.h"
```

Macros

- #define SYNAPSEMODELS_CC

Functions

- void prepareWeightUpdateModels ()

  Function that prepares the standard (pre) synaptic models, including their variables, parameters, dependent parameters and code strings.

Variables

- vector< weightUpdateModel > weightUpdateModels

  Global C++ vector containing all weightupdate model descriptions.
- unsigned int NSYNAPSE

  Variable attaching the name NSYNAPSE to the non-learning synapse.
- unsigned int NGRADSYNAPSE

  Variable attaching the name NGRADSYNAPSE to the graded synapse wrt the presynaptic voltage.
- unsigned int LEARN1SYNAPSE

  Variable attaching the name LEARN1SYNAPSE to the the primitive STDP model for learning.

19.65.1 Macro Definition Documentation

19.65.1.1 #define SYNAPSEMODELS_CC

19.65.2 Function Documentation

19.65.2.1 void prepareWeightUpdateModels ( )

Function that prepares the standard (pre) synaptic models, including their variables, parameters, dependent parameters and code strings.

19.65.3 Variable Documentation

19.65.3.1 unsigned int LEARN1SYNAPSE

Variable attaching the name LEARN1SYNAPSE to the the primitive STDP model for learning.

19.65.3.2 unsigned int NGRADSYNAPSE

Variable attaching the name NGRADSYNAPSE to the graded synapse wrt the presynaptic voltage.

19.65.3.3 unsigned int NSYNAPSE

Variable attaching the name NSYNAPSE to the non-learning synapse.

19.65.3.4 vector<weightUpdateModel> weightUpdateModels

Global C++ vector containing all weightupdate model descriptions.

19.66 synapseModels.h File Reference

```
#include "dpclass.h"
#include <string>
#include <vector>
```

Classes

- class weightUpdateModel

  Class to hold the information that defines a weightupdate model (a model of how spikes affect synaptic (and/or) (mostly) post-synaptic neuron variables. It also allows to define changes in response to post-synaptic spikes/spike-like events.
- class pwSTDP

  TODO This class definition may be code-generated in a future release.

Functions

- void prepareWeightUpdateModels ()

  Function that prepares the standard (pre) synaptic models, including their variables, parameters, dependent parameters and code strings.

Variables

- vector< weightUpdateModel > weightUpdateModels

  Global C++ vector containing all weightupdate model descriptions.
- unsigned int NSYNAPSE

  Variable attaching the name NSYNAPSE to the non-learning synapse.
- unsigned int NGRADSYNAPSE

  Variable attaching the name NGRADSYNAPSE to the graded synapse wrt the presynaptic voltage.
- unsigned int LEARN1SYNAPSE

  Variable attaching the name LEARN1SYNAPSE to the the primitive STDP model for learning.
- const unsigned int SYNTYPENO = 4

19.66.1 Function Documentation

19.66.1.1 void prepareWeightUpdateModels ( )

Function that prepares the standard (pre) synaptic models, including their variables, parameters, dependent parameters and code strings.

19.66.2 Variable Documentation

19.66.2.1 unsigned int LEARN1SYNAPSE

Variable attaching the name LEARN1SYNAPSE to the the primitive STDP model for learning.

19.66.2.2 unsigned int NGRADSYNAPSE

Variable attaching the name NGRADSYNAPSE to the graded synapse wrt the presynaptic voltage.

19.66.2.3 unsigned int NSYNAPSE

Variable attaching the name NSYNAPSE to the non-learning synapse.

19.66.2.4 const unsigned int SYNTYPENO = 4

19.66.2.5 vector<weightUpdateModel> weightUpdateModels

Global C++ vector containing all weightupdate model descriptions.

19.67 utils.cc File Reference

```
#include "utils.h"
#include <fstream>
#include <cstdint>
#include "codeStream.h"
```

Macros

- #define UTILS_CC

Functions

- CUresult cudaFuncGetAttributesDriver (cudaFuncAttributes ∗attr, CUfunction kern)

  Function for getting the capabilities of a CUDA device via the driver API.
- void writeHeader (CodeStream &os)

  Function to write the comment header denoting file authorship and contact details into the generated code.
- unsigned int theSize (const string &type)

  Tool for determining the size of variable types on the current architecture.

19.67.1 Macro Definition Documentation

19.67.1.1 #define UTILS_CC

19.67.2 Function Documentation

19.67.2.1 CUresult cudaFuncGetAttributesDriver (cudaFuncAttributes ∗ *attr,* CUfunction *kern*)

Function for getting the capabilities of a CUDA device via the driver API.

19.67.2.2 unsigned int theSize (const string & *type*)

Tool for determining the size of variable types on the current architecture.

19.67.2.3 void writeHeader (CodeStream & *os*)

Function to write the comment header denoting file authorship and contact details into the generated code.

19.68 utils.h File Reference

This file contains standard utility functions provide within the NVIDIA CUDA software development toolkit (SDK). The remainder of the file contains a function that defines the standard neuron models.

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <cuda.h>
#include <cuda_runtime.h>
```

Macros

- #define _UTILS_H_

  macro for avoiding multiple inclusion during compilation
- #define CHECK_CU_ERRORS(call) call

  Macros for catching errors returned by the CUDA driver and runtime APIs.
- #define CHECK_CUDA_ERRORS(call)
- #define B(x, i) ((x) & (0x80000000 >> (i)))

  Bit tool macros.
- #define setB(x, i) x= ((x) | (0x80000000 >> (i)))

  Set the bit at the specified position i in x to 1.
- #define delB(x, i) x= ((x) & (∼(0x80000000 >> (i))))

  Set the bit at the specified position i in x to 0.
- #define USE(expr) do { (void)(expr); } while (0)

  Miscellaneous macros.

Functions

- CUresult cudaFuncGetAttributesDriver (cudaFuncAttributes ∗attr, CUfunction kern)

  Function for getting the capabilities of a CUDA device via the driver API.
- void gennError (const string &error)

  Function called upon the detection of an error. Outputs an error message and then exits.
- unsigned int theSize (const string &type)

  Tool for determining the size of variable types on the current architecture.
- void writeHeader (CodeStream &os)

  Function to write the comment header denoting file authorship and contact details into the generated code.

19.68.1 Detailed Description

This file contains standard utility functions provide within the NVIDIA CUDA software development toolkit (SDK). The remainder of the file contains a function that defines the standard neuron models.

19.68.2 Macro Definition Documentation

19.68.2.1 #define _UTILS_H_

macro for avoiding multiple inclusion during compilation

19.68.2.2 #define B(*x, i*) ((x) & (0x80000000 >> (i)))

Bit tool macros.

Extract the bit at the specified position i from x

19.68.2.3 #define CHECK_CU_ERRORS(*call*) call

Macros for catching errors returned by the CUDA driver and runtime APIs.

19.68.2.4 #define CHECK_CUDA_ERRORS(*call*)

Value:

```
{                                                                  \
    cudaError_t error = call;                                          \
    if (error != cudaSuccess)                                          \
      {                                                                  \
        cerr << __FILE__ << ": " <<  __LINE__;                           \
        cerr << ": cuda runtime error " << error << ": ";            \
        cerr << cudaGetErrorString(error) << endl;                    \
        exit(EXIT_FAILURE);                                           \
      }                                                                  \
  }
```

19.68.2.5 #define delB(*x*, *i*) x= ((x) & ($\sim$(0x80000000 $>>$ (i))))

Set the bit at the specified position i in x to 0.

19.68.2.6 #define setB(*x*, *i*) x= ((x) | (0x80000000 $>>$ (i)))

Set the bit at the specified position i in x to 1.

19.68.2.7 #define USE(*expr*) do { (void)(expr); } while (0)

Miscellaneous macros.

Silence 'unused parameter' warnings

19.68.3 Function Documentation

19.68.3.1 CUresult cudaFuncGetAttributesDriver (cudaFuncAttributes ∗ *attr*, CUfunction *kern*)

Function for getting the capabilities of a CUDA device via the driver API.

19.68.3.2 void gennError (const string & *error*) [inline]

Function called upon the detection of an error. Outputs an error message and then exits.

19.68.3.3 unsigned int theSize (const string & *type*)

Tool for determining the size of variable types on the current architecture.

19.68.3.4 void writeHeader (CodeStream & *os*)

Function to write the comment header denoting file authorship and contact details into the generated code.

19.69 variableMode.h File Reference

```
#include <cstdint>
```

Enumerations

- enum VarLocation : uint8_t { VarLocation::HOST = (1 $<<$ 0), VarLocation::DEVICE = (1 $<<$ 1), VarLocation↩ ::ZERO_COPY = (1 $<<$ 2) }

  $<$ Flags defining which memory space variables should be allocated in

- enum VarInit : uint8_t { VarInit::HOST = (1 << 3), VarInit::DEVICE = (1 << 4) }
- enum VarMode : uint8_t {
  VarMode::LOC_DEVICE_INIT_DEVICE = static_cast<uint8_t>(VarLocation::DEVICE) | static_cast<uint8_t>(VarInit::DEVICE), VarMode::LOC_HOST_DEVICE_INIT_HOST = static_cast<uint8_t>(VarLocation::HOST) | static_cast<uint8_t>(VarLocation::DEVICE) | static_cast<uint8_t>(VarInit::HOST), VarMode::LOC_HOST_DEVICE_INIT_DEVICE = static_cast<uint8_t>(VarLocation::HOST) | static_cast<uint8_t>(VarLocation::DEVICE) | static_cast<uint8_t>(VarInit::DEVICE), VarMode::LOC_ZERO_COPY_INIT_HOST = static_cast<uint8_t>(VarLocation::HOST) | static_cast<uint8_t>(VarLocation::DEVICE) | static_cast<uint8_t>(VarLocation::ZERO_COPY) | static_cast<uint8_t>(VarInit::HOST),
  VarMode::LOC_ZERO_COPY_INIT_DEVICE = static_cast<uint8_t>(VarLocation::HOST) | static_cast<uint8_t>(VarLocation::DEVICE) | static_cast<uint8_t>(VarLocation::ZERO_COPY) | static_cast<uint8_t>(VarInit::DEVICE) }

Functions

- bool operator& (VarMode mode, VarInit init)
- bool operator& (VarMode mode, VarLocation location)

19.69.1 Enumeration Type Documentation

19.69.1.1 enum **VarInit** : uint8_t [strong]

Enumerator

HOST

DEVICE

19.69.1.2 enum **VarLocation** : uint8_t [strong]

< Flags defining which memory space variables should be allocated in

Enumerator

HOST

DEVICE

ZERO_COPY

19.69.1.3 enum **VarMode** : uint8_t [strong]

Enumerator

LOC_DEVICE_INIT_DEVICE

LOC_HOST_DEVICE_INIT_HOST

LOC_HOST_DEVICE_INIT_DEVICE

LOC_ZERO_COPY_INIT_HOST

LOC_ZERO_COPY_INIT_DEVICE

19.69.2 Function Documentation

19.69.2.1 bool operator& (**VarMode** *mode*, **VarInit** *init*) [inline]

19.69.2.2 bool operator& (**VarMode** *mode*, **VarLocation** *location*) [inline]

References

[1] Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003. 8, 9, 10, 11, 51, 58, 80, 81, 82, 83, 185

[2] T. Nowotny. Parallel implementation of a spiking neuronal network model of unsupervised olfactory learning on NVidia CUDA. In P. Sobrevilla, editor, *IEEE World Congress on Computational Intelligence*, pages 3238–3245, Barcelona, 2010. IEEE. 29

[3] Thomas Nowotny, Ramón Huerta, Henry DI Abarbanel, and Mikhail I Rabinovich. Self-organization in the olfactory system: one shot odor recognition in insects. *Biological cybernetics*, 93(6):436–446, 2005. 12, 124

[4] Nikolai F Rulkov. Modeling of spiking-bursting neural behavior using two-dimensional map. *Physical Review E*, 65(4):041922, 2002. 124

[5] R. D. Traub and R. Miles. *Neural Networks of the Hippocampus*. Cambridge University Press, New York, 1991. 12, 13, 137

Index