

---

# **Brian2GeNN Documentation**

*Release 1.0*

**Brian2GeNN authors**

**Aug 21, 2017**



<b>1</b>	<b>Using Brian2GeNN</b>	<b>3</b>
1.1	Installing the Brian2GeNN interface . . . . .	3
1.2	Using the Brian2GeNN interface . . . . .	4
<b>2</b>	<b>Unsupported features in Brian2GeNN</b>	<b>5</b>
2.1	Summed variables . . . . .	5
2.2	Linked variables . . . . .	5
2.3	Custom events . . . . .	6
2.4	Heterogeneous delays . . . . .	6
2.5	Multiple synaptic pathways . . . . .	7
2.6	Timed arrays . . . . .	7
2.7	Multiple clocks . . . . .	7
2.8	Multiple runs . . . . .	7
2.9	Multiple networks . . . . .	7
2.10	Custom schedules . . . . .	7
<b>3</b>	<b>Brian2GeNN specific preferences</b>	<b>9</b>
3.1	Connectivity . . . . .	9
3.2	Compiler preferences . . . . .	9
3.3	List of preferences . . . . .	10
<b>4</b>	<b>How Brian2GeNN works inside</b>	<b>11</b>
4.1	Model and user code in GeNN . . . . .	11
4.2	Code generation pipeline in Brian2GeNN . . . . .	11
4.3	Templates in Brian2GeNN . . . . .	12
4.4	Data transfers and results . . . . .	12
4.5	Memory usage . . . . .	12
<b>5</b>	<b>brian2genn package</b>	<b>13</b>
5.1	binomial module . . . . .	13
5.2	codeobject module . . . . .	13
5.3	correctness_testing module . . . . .	14
5.4	device module . . . . .	15
5.5	genn_generator module . . . . .	22
5.6	insyn module . . . . .	24
5.7	preferences module . . . . .	25
5.8	Subpackages . . . . .	25

<b>6 Indices and tables</b>	<b>27</b>
<b>Python Module Index</b>	<b>29</b>

Contents:



---

## Using Brian2GeNN

---

Brian supports generating standalone code for multiple devices. In this mode, running a Brian script generates source code in a project tree for the target device/language. This code can then be compiled and run on the device, and modified if needed. The Brian2GeNN package provides such a ‘device’ to run **Brian 2** code on the **GeNN** (GPU enhanced Neuronal Networks) backend. GeNN is in itself a code-generation based framework to generate and execute code for NVIDIA CUDA. Through Brian2GeNN one can hence generate and run CUDA code on NVIDIA GPUs based solely in Brian 2 input.

### Installing the Brian2GeNN interface

In order to use the Brian2GeNN interface, all three Brian 2, GeNN and Brian2GeNN need to be fully installed. The easiest way to do this is by using the `conda` package provided in the `brian-team` channel on <https://anaconda.org>. This will install Brian 2 and its dependencies, and Brian2GeNN with an internal version of GeNN (you can always switch to using an existing GeNN installation by setting the `devices.genn.path` preference). Note that this will *not* install the CUDA toolkit and driver necessary to run simulations on a NVIDIA graphics card. These will have to be installed manually, e.g. from [NVIDIA’s web site](#) (you can always run simulations in the “CPU-only” mode, but that of course defeats the main purpose of Brian2GeNN...). Depending on the installation method, you might also have to manually set the `CUDA_PATH` environment variable (or alternatively the `devices.genn.cuda_path` preference) to point to CUDA’s installation directory.

To install Brian2GeNN via conda use:

```
conda install -c brian-team brian2genn
```

If you are not using the conda package manager or if there is no conda package for your architecture, you can always install `brian2genn` from its source package on <http://pypi.python.org/>

```
pip install brian2genn
```

(might require administrator privileges depending on the configuration of your system; add `--user` to force an installation with user privileges only). Note that in this case, GeNN needs to be installed manually (see its [installation instructions](#)), and either the `GENN_PATH` environment variable of the `devices.genn.path` preference have to point to its directory. In addition, the CUDA libraries have to be installed (see above).

**Note:** The above commands install the necessary packages to run simulations with Brian2/GeNN, but most users would install additional packages, e.g. `matplotlib` for plotting. This can be done with the same package management tools mentioned above, e.g. `conda install matplotlib` or `pip install matplotlib`.

---

## Using the Brian2GeNN interface

To use the interface one then needs to import the `brian2genn` interface:

```
import brian2genn
```

Then you need to choose the ‘genn’ device at the beginning of the Brian 2 script, i.e. after the import statements, add:

```
set_device('genn')
```

At the encounter of the first `run` statement (Brian2GeNN does currently only support a single `run` statement per script), code for GeNN will be generated, compiled and executed.

The `set_device` function can also take additional arguments, e.g. to run GeNN in its “CPU-only” mode and to get additional debugging output, use:

```
set_device('genn', useGPU=False, debug=True)
```

Not all features of Brian work with Brian2GeNN. The current list of excluded features is detailed in *Unsupported features in Brian2GeNN*.



---

## Unsupported features in Brian2GeNN

---

### Summed variables

Summed variables are currently not supported in GeNN due to the cross- population nature of this feature. However, a simple form of summed variable is supported and intrinsic to GeNN. This is the action of ‘pre’ code in a `Synapses` definition onto a pre-synaptic variable. The allowed interaction is summing onto one pre-synaptic variable from each `Synapses` group.

### Linked variables

Linked variables create a communication overhead that is problematic in GeNN. They are therefore at the moment not supported. In principle support for this feature could be added but in the meantime we suggest to look into avoiding linked variables by combining groups that are linked. For example

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

# Common deterministic input
N = 25
tau_input = 5*ms
input = NeuronGroup(N, 'dx/dt = -x / tau_input + sin(0.1*t/ tau_input) : 1')

# The noisy neurons receiving the same input
tau = 10*ms
sigma = .015
eqs_neurons = '''
dx/dt = (0.9 + .5 * I - x) / tau + sigma * (2 / tau)**.5 * xi : 1
I : 1 (linked)
'''
neurons = NeuronGroup(N, model=eqs_neurons, threshold='x > 1',
                      reset='x = 0', refractory=5*ms)
```

```
neurons.x = 'rand()'
neurons.I = linked_var(input, 'x') # input.x is continuously fed into neurons.I
spikes = SpikeMonitor(neurons)

run(500*ms)example
```

could be replaced by

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

N = 25
tau_input = 5*ms

# Noisy neurons receiving the same deterministic input
tau = 10*ms
sigma = .015
eqs_neurons = '''
dI/dt= -I / tau_input + sin(0.1*t/ tau_input) : 1'
dx/dt = (0.9 + .5 * I - x) / tau + sigma * (2 / tau)**.5 * xi : 1
'''
neurons = NeuronGroup(N, model=eqs_neurons, threshold='x > 1',
                      reset='x = 0', refractory=5*ms)
neurons.x = 'rand()'
spikes = SpikeMonitor(neurons)

run(500*ms)example
```

In this second solution the variable `I` is calculated multiple times within the ‘noisy neurons’, which in a sense is an unnecessary computational overhead. However, in the massively parallel GPU accelerators this is not necessarily a problem. Note that this method only works where the common input is deterministic. If the input had been:

```
input = NeuronGroup(1, 'dx/dt = -x / tau_input + (2 /tau_input)**.5 * xi : 1')
```

i.e. contains a random element, then moving the common input into the ‘noisy neuron’ population would make it individual, independent noisy inputs with likely quite different results.

## Custom events

GeNN does not support custom event types in addition to the standard threshold and reset, they can therefore not be used with the Brian2GeNN backend.

## Heterogeneous delays

At the moment, GeNN only has support for a single homogeneous delay for each synaptic population. Brian simulations that use heterogeneous delays can therefore not use the Brian2GeNN backend. In simple cases with just a few different delay values (e.g. one set of connections with a short and another set of connections with a long delay), this limitation can be worked around by creating multiple `Synapses` objects with each using a homogeneous delay.

## Multiple synaptic pathways

GeNN does not have support for multiple synaptic pathways as Brian 2 does, you can therefore only use a single `pre` and `post` pathway with Brian2GeNN.

## Timed arrays

Timed arrays pose a problem in the Brian2GeNN interface because they necessitate communication from the timed array to the target group at runtime that would result in host to GPU copies in the final CUDA/C++ code. This could lead to large inefficiencies and for the moment we have therefore decided to not support this feature.

## Multiple clocks

GeNN is by design operated with a single clock with a fixed time step across the entire simulation. If you are using multiple clocks and they are commensurate, please reformulate your script using just the fastest clock as the standard clock. If your clocks are not commensurate, and this is essential for your simulation, Brian2GeNN can unfortunately not be used.

## Multiple runs

GeNN is designed for single runs and cannot be used for the Brian style multiple runs. However, if this is of use, code can be run repeatedly “in multiple runs” that are completely independent. This just needs a `reset_device` command issued after the `run(runtime)` command. Note, however, that these multiple runs are completely independent, i.e. for the second run the code generation pipeline for Brian2GeNN is repeated in its entirety which may incur a measurable delay.

## Multiple networks

Multiple networks cannot be supported in the Brian2GeNN interface. Please use only a single network, either by creating it explicitly as a `Network` object or by not creating any (i.e. using Brian’s “magic” system).

## Custom schedules

GeNN has a fixed order of operations during a time step, Brian’s more flexible scheduling model (e.g. changing a network’s schedule or individual objects’ `when` attribute) can therefore not be used.



---

## Brian2GeNN specific preferences

---

### Connectivity

The preference `devices.genn.connectivity` determines what connectivity scheme is used within GeNN to represent the connections between neurons. GeNN supports the use of full connectivity matrices ('DENSE') or a representation where connections are represented with sparse matrix methods ('SPARSE'). You can set the preference like this:

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

prefs.devices.genn.connectivity = 'DENSE'
```

### Compiler preferences

Brian2GeNN will use the compiler preferences specified for Brian2 for the C++ compiler call. This means you should set the `codegen.cpp.extra_compile_args` preference, or set `codegen.cpp.extra_compile_args_gcc` and `codegen.cpp.extra_compile_args_msvc` to set preferences specifically for compilation under Linux/OS-X and Windows, respectively.

Brian2GeNN also offers a preference to specify additional compiler flags for the CUDA compilation with the `nvcc` compiler: `devices.genn.extra_compile_args_nvcc`.

Note that all of the above preferences expect a *Python list* of individual compiler arguments, i.e. to for example add an argument for the `nvcc` compiler, use:

```
prefs.devices.genn.extra_compile_args_nvcc += ['--verbose']
```

On Windows, Brian2GeNN will try to find the file `vcvarsall.bat` to enable compilation with the MSVC compiler automatically. If this fails, or if you have multiple versions of MSVC installed and want to select a specific one, you can set the `codegen.cpp.msvc_vars_location` preference.

## List of preferences

Preferences that relate to the brian2genn interface

**devices.genn.auto\_choose\_device = True** The GeNN preference autoChooseDevice that determines whether or not a GPU should be chosen automatically when multiple CUDA enabled devices are present.

`devices.genn.connectivity = 'SPARSE'`

This preference determines which connectivity scheme is to be employed within GeNN. The valid alternatives are 'DENSE' and 'SPARSE'. For 'DENSE' the GeNN dense matrix methods are used for all connectivity matrices. When 'SPARSE' is chosen, the GeNN sparse matrix representations are used.

**devices.genn.cuda\_path = None** The path to the CUDA installation (if not set, the CUDA\_PATH environment variable will be used instead)

**devices.genn.default\_device = 0** The GeNN preference defaultDevice that determines CUDA enabled device should be used if it is not automatically chosen.

**devices.genn.extra\_compile\_args\_nvcc = ['-O3']** Extra compile arguments (a list of strings) to pass to the nvcc compiler.

**devices.genn.path = None** The path to the GeNN installation (if not set, the GENN\_PATH environment variable will be used instead)

---

## How Brian2GeNN works inside

---

The Brian2GeNN interface is providing middleware to use the GeNN simulator framework as a backend to the Brian 2 simulator. It has been designed in a way that makes maximal use of the existing Brian 2 code base by deriving large parts of the generated code from the `cpp_standalone` device of Brian 2.

### Model and user code in GeNN

In GeNN a simulation is assembled from two main sources of code. Users of GeNN provide “code snippets” as C++ strings that define neuron and synapse models. These are then assembled into neuronal networks in a model definition function. Based on the model definition, GeNN generates GPU and equivalent CPU simulation code for the described network. This is the first source of code.

The actual simulation and handling input and output data is the responsibility of the user in GeNN. Users provide their own C/C++ code for this that utilizes the generated code described above for the core simulation but is otherwise fully independent of the core GeNN system.

In the Brian2GeNN both the model definition and the user code for the main simulation are derived from the Brian 2 model description. The user side code for data handling etc derives more or less directly from the Brian 2 `cpp_standalone` device in the form of `GennUserCodeObjects`. The model definition code and “code snippets” derive from separate templates and are encapsulated into `GeNNCodeObjects`.

### Code generation pipeline in Brian2GeNN

The model generation pipeline in Brian2GeNN involves a number of steps. First, Brian 2 performs the usual interpretation of equations and unit checking, as well as, applying an integration scheme onto ODEs. The resulting abstract code is then translated into C++ code for `GeNNUserCodeObjects` and C++-like code for `GeNNCodeObjects`. These are then assembled using templating in Jinja2 into C++ code and GeNN model definition code. The details of making Brian 2’s `cpp_standalone` code suitable for the GeNN user code and GeNN model definition code and code snippets are taken care of in the `GeNNDevice.build` function.

Once all the sources have been generated, the resulting GeNN project is built with the GeNN code generation pipeline. See the GeNN manual for more details on this process.

## Templates in Brian2GeNN

The templates used for code generation in Brian2GeNN, as mentioned above, partially derive from the `cpp_standalone` templates of Brian 2. More than half of the templates are identical. Other templates, however, in particular for the model definition file and the main simulation engine and main entry file “`runner.cc`” have been specifically written for Brian2GeNN to produce a valid GeNN project.

## Data transfers and results

In Brian 2, data structures for initial values and synaptic connectivities etc are written to disk into binary files if a standalone device is used. The executable of the standalone device then reads the data from disk and initializes its variables with it. In Brian2GeNN the same mechanism is used, and after the data has been read from disk with the native `cpp_standalone` methods, there is a translation step, where Brian2GeNN provides code that translates the data from `cpp_standalone` arrays into the appropriate GeNN data structures. The methods for this process are provided in the static (not code-generated) “`b2glib`”.

At the end of a simulation, the inverse process takes place and GeNN data is transferred back into `cpp_standalone` arrays. Native Brian 2 `cpp_standalone` code is then invoked to write data back to disk.

If monitors are used, the translation occurs at every instance when monitors are updated.

## Memory usage

Related to the implementation of data flows in Brian2GeNN described above the host memory used in a run in brian2GeNN is about twice what would have been used in a Brian 2 native `cpp_standalone` implementation because all data is held in two different formats - as `cpp_standalone` arrays and as GeNN data structures.



## binomial module

Implementation of `BinomialFunction`

## codeobject module

Brian2GeNN defines two different types of code objects, `GeNNCodeObject` and `GeNNUserCodeObject`. `GeNNCodeObject` is the class of code objects that produce code snippets for GeNN neuron or synapse models. `GeNNUserCodeObject` is the class of code objects that produce C++ code which is used as “user-side” code in GeNN. The class derives directly from Brian 2’s `CPPStandaloneCodeObject`, using the `CPPCodeGenerator`.

*Exported members:* `GeNNCodeObject`, `GeNNUserCodeObject`

### Classes

<code>GeNNCodeObject(owner, code, variables, ...)</code>	Class of code objects that generate GeNN “code snippets”
--	--

## GeNNCodeObject class

(Shortest import: `from brian2genn.codeobject import GeNNCodeObject`)

```
class brian2genn.codeobject.GeNNCodeObject(owner, code, variables, variable_indices,
                                           template_name,           template_source,
                                           name='codeobject*')
```

Bases: `brian2.codegen.codeobject.CodeObject`

Class of code objects that generate GeNN “code snippets”

<code>GeNNUserCodeObject(owner, code, variables, ...)</code>	Class of code objects that generate GeNN “user code”
--	--

## GeNNUserCodeObject class

(Shortest import: `from brian2genn.codeobject import GeNNUserCodeObject`)

```
class brian2genn.codeobject.GeNNUserCodeObject (owner, code, variables, variable_indices,  
                                              template_name,          template_source,  
                                              name='codeobject*')
```

Bases: `brian2.devices.cpp_standalone.codeobject.CPPStandaloneCodeObject`

Class of code objects that generate GeNN “user code”

## correctness\_testing module

Definitions of the configuration for correctness testing.

Exported members: `GeNNConfiguration`, `GeNNConfigurationCPU`, `GeNNConfigurationOptimized`

### Classes

---

`GeNNConfiguration([maximum_run_time])`

### Methods

---

## GeNNConfiguration class

(Shortest import: `from brian2genn.correctness_testing import GeNNConfiguration`)

```
class brian2genn.correctness_testing.GeNNConfiguration (maximum_run_time=10. *  
                                                       Msecond)
```

Bases: `brian2.tests.features.base.Configuration`

### Methods

---

`before_run()`

---

### Details

`before_run ()`

---

`GeNNConfigurationCPU([maximum_run_time])`

### Methods

---

## GeNNConfigurationCPU class

(Shortest import: `from brian2genn.correctness_testing import GeNNConfigurationCPU`)

```
class brian2genn.correctness_testing.GeNNConfigurationCPU (maximum_run_time=10. *  
                                                           Msecond)
```

Bases: `brian2.tests.features.base.Configuration`

## Methods

---

`before_run()`

---

## Details

`before_run()`

---

`GeNNConfigurationOptimized([maximum_run_time])`

## Methods

---

## GeNNConfigurationOptimized class

(Shortest `import:` `from brian2genn.correctness_testing import GeNNConfigurationOptimized`)

`class brian2genn.correctness_testing.GeNNConfigurationOptimized (maximum_run_time=10. * Msecond)`

Bases: `brian2.tests.features.base.Configuration`

## Methods

---

`before_run()`

---

## Details

`before_run()`

## device module

Module implementing the bulk of the `brian2genn` interface by defining the “genn” device.

*Exported members:* `GeNNDevice`

## Classes

---

<code>CPPWriter(project_dir)</code>	Class that provides the method for writing C++ files from a string of code.
-------------------------------------	---

---

## CPPWriter class

(Shortest `import:` `from brian2genn.device import CPPWriter`)

**class** `brian2genn.device.CPPWriter` (*project\_dir*)

Bases: `object`

Class that provides the method for writing C++ files from a string of code.

### Methods

---

`write`(filename, contents)

---

### Details

**write** (*filename, contents*)

---

`DelayedCodeObject`(owner, name, ...)

Dummy class used for delaying the `CodeObject` creation of stateupdater, thresholder, and resetter of a `NeuronGroup` (which will all be merged into a single code object).

---

## DelayedCodeObject class

(Shortest import: `from brian2genn.device import DelayedCodeObject`)

**class** `brian2genn.device.DelayedCodeObject` (*owner, name, abstract\_code, variables, variable\_indices, override\_conditional\_write*)

Bases: `object`

Dummy class used for delaying the `CodeObject` creation of stateupdater, thresholder, and resetter of a `NeuronGroup` (which will all be merged into a single code object).

---

`GeNNDevice`()

The main “genn” device.

---

## GeNNDevice class

(Shortest import: `from brian2genn.device import GeNNDevice`)

**class** `brian2genn.device.GeNNDevice`

Bases: `brian2.devices.cpp_standalone.device.CPPStandaloneDevice`

The main “genn” device. This does most of the translation work from Brian 2 generated code to functional GeNN code, assisted by the “GeNN language”.

### Attributes

---

`source_files`

List of all source and header files (to be included in runner)

---

### Methods

---

`activate`([*build\_on\_run*])

---

Continued on next page

Table 5.14 – continued from previous page

<code>add_array_variable(model, varname, variable)</code>	
<code>add_array_variables(model, owner)</code>	
<code>add_parameter(model, varname, variable)</code>	
<code>build([directory, compile, run, use_GPU, ...])</code>	This function does the main post-translation work for the <code>genn</code> device.
<code>code_object(owner, name, abstract_code, ...)</code>	Processes abstract code into code objects and stores them in different arrays for <code>GeNNCodeObjects</code> and <code>GeNNUserCodeObjects</code> .
<code>code_object_class([codeobj_class])</code>	
<code>collect_synapses_variables(synapse_model, ...)</code>	
<code>compile_source(debug, directory, use_GPU)</code>	
<code>copy_source_files(writer, directory)</code>	
<code>fill_with_array(var, arr)</code>	
<code>fix_random_generators(model, code)</code>	Translates <code>cpp_standalone</code> style random number generator calls into GeNN-compatible calls by replacing the <code>cpp_standalone_vectorisation_idx</code> argument with the <code>GeNN_seed</code> argument.
<code>fix_synapses_code(synapse_model, pathway, ...)</code>	
<code>generate_code_objects(writer)</code>	
<code>generate_engine_source(writer)</code>	
<code>generate_main_source(writer, main_lines)</code>	
<code>generate_makefile(directory, use_GPU)</code>	
<code>generate_model_source(writer)</code>	
<code>generate_objects_source(arange_arrays, net, ...)</code>	
<code>make_main_lines()</code>	Generates the code lines that handle initialisation of Brian 2 <code>cpp_standalone</code> type arrays.
<code>network_run(net, duration[, report, ...])</code>	
<code>process_neuron_groups(neuron_groups, objects)</code>	
<code>process_poisson_groups(objects, poisson_groups)</code>	
<code>process_rate_monitors(rate_monitors)</code>	
<code>process_spike_monitors(spike_monitors)</code>	
<code>process_spikegenerators(spikegenerator_groups)</code>	
<code>process_state_monitors(directory, ...)</code>	
<code>process_synapses(synapse_groups)</code>	
<code>run(directory, use_GPU, with_output)</code>	
<code>variableview_set_with_expression(...[, ...])</code>	
<code>variableview_set_with_expression_conditional(...)</code>	
<code>variableview_set_with_index_array(...)</code>	

## Details

### **source\_files**

List of all source and header files (to be included in runner)

**activate** (*build\_on\_run=True, \*\*kwargs*)

**add\_array\_variable** (*model, varname, variable*)

**add\_array\_variables** (*model, owner*)

**add\_parameter** (*model, varname, variable*)

**build** (*directory='GeNNworkspace', compile=True, run=True, use\_GPU=True, debug=False, with\_output=True, direct\_call=True*)

This function does the main post-translation work for the `genn` device. It uses the code generated during/before `run()` and extracts information about neuron groups, synapse groups, monitors, etc. that is then formatted for use in GeNN-specific templates. The overarching strategy of the `brian2genn` interface is to use `cpp_standalone` code generation and templates for most of the “user-side code” (in the meaning defined in GeNN) and have GeNN-specific templates for the model definition and the main code for the executable that pulls everything together (in `main.cpp` and `engine.cpp` templates). The handling of input/output arrays for everything is lent from `cpp_standalone` and the `cpp_standalone` arrays are then translated into GeNN-suitable data structures using the static (not code-generated) `b2glib` library functions. This means that the GeNN specific code only has to be concerned about executing the correct model and feeding back results into the appropriate `cpp_standalone` data structures.

**code\_object** (*owner, name, abstract\_code, variables, template\_name, variable\_indices, codeobj\_class=None, template\_kwds=None, override\_conditional\_write=None*)

Processes abstract code into code objects and stores them in different arrays for `GeNNCodeObjects` and `GeNNUserCodeObjects`.

**code\_object\_class** (*codeobj\_class=None, \*args, \*\*kwds*)

**collect\_synapses\_variables** (*synapse\_model, pathway, codeobj*)

**compile\_source** (*debug, directory, use\_GPU*)

**copy\_source\_files** (*writer, directory*)

**fill\_with\_array** (*var, arr*)

**fix\_random\_generators** (*model, code*)

Translates `cpp_standalone` style random number generator calls into GeNN-compatible calls by replacing the `cpp_standalone_vectorisation_idx` argument with the `GeNN_seed` argument.

**fix\_synapses\_code** (*synapse\_model, pathway, codeobj, code*)

**generate\_code\_objects** (*writer*)

**generate\_engine\_source** (*writer*)

**generate\_main\_source** (*writer, main\_lines*)

**generate\_makefile** (*directory, use\_GPU*)

**generate\_model\_source** (*writer*)

**generate\_objects\_source** (*arange\_arrays, net, static\_array\_specs, synapses, writer*)

**make\_main\_lines** ()

Generates the code lines that handle initialisation of Brian 2 `cpp_standalone` type arrays. These are then translated into the appropriate GeNN data structures in separately generated code.

**network\_run** (*net, duration, report=None, report\_period=10. \* second, namespace=None, profile=False, level=0, \*\*kwds*)

**process\_neuron\_groups** (*neuron\_groups, objects*)

**process\_poisson\_groups** (*objects, poisson\_groups*)

**process\_rate\_monitors** (*rate\_monitors*)

**process\_spike\_monitors** (*spike\_monitors*)

**process\_spikegenerators** (*spikegenerator\_groups*)

**process\_state\_monitors** (*directory, state\_monitors, writer*)

**process\_synapses** (*synapse\_groups*)

**run** (*directory, use\_GPU, with\_output*)

**variableview\_set\_with\_expression** (*variableview, item, code, run\_namespace, check\_units=True*)

**variableview\_set\_with\_expression\_conditional** (*variableview, cond, code, run\_namespace, check\_units=True*)

**variableview\_set\_with\_index\_array** (*variableview, item, value, check\_units*)

---

<code>neuronModel()</code>	Class that contains all relevant information of a neuron model.
----------------------------	---

---

## neuronModel class

(Shortest import: `from brian2genn.device import neuronModel`)

**class** `brian2genn.device.neuronModel`

Bases: `object`

Class that contains all relevant information of a neuron model.

---

<code>rateMonitorModel()</code>	Class that contains all relevant information about a rate monitor.
---------------------------------	--

---

## rateMonitorModel class

(Shortest import: `from brian2genn.device import rateMonitorModel`)

**class** `brian2genn.device.rateMonitorModel`

Bases: `object`

Class that contains all relevant information about a rate monitor.

---

<code>spikeMonitorModel()</code>	Class the contains all relevant information about a spike monitor.
----------------------------------	--

---

## spikeMonitorModel class

(Shortest import: `from brian2genn.device import spikeMonitorModel`)

**class** `brian2genn.device.spikeMonitorModel`

Bases: `object`

Class the contains all relevant information about a spike monitor.

---

<code>spikegeneratorModel()</code>	Class that contains all relevant information of a spike generator group.
------------------------------------	--

---

## spikegeneratorModel class

(Shortest import: `from brian2genn.device import spikegeneratorModel`)

**class** `brian2genn.device.spikegeneratorModel`

Bases: `object`

Class that contains all relevant information of a spike generator group.

---

`stateMonitorModel()`

Class that contains all relevant information about a state monitor.

---

## stateMonitorModel class

(Shortest import: `from brian2genn.device import stateMonitorModel`)

**class** `brian2genn.device.stateMonitorModel`

Bases: `object`

Class that contains all relevant information about a state monitor.

---

`synapseModel()`

Class that contains all relevant information about a synapse model.

---

## synapseModel class

(Shortest import: `from brian2genn.device import synapseModel`)

**class** `brian2genn.device.synapseModel`

Bases: `object`

Class that contains all relevant information about a synapse model.

## Functions

---

`decorate`(code, variables, shared\_variables, ...)

Support function for inserting GeNN-specific “decorations” for variables and parameters, such as `$(.)`.

---

## decorate function

(Shortest import: `from brian2genn.device import decorate`)

`brian2genn.device.decorate` (code, variables, shared\_variables, parameters, do\_final=True)

Support function for inserting GeNN-specific “decorations” for variables and parameters, such as `$(.)`.

---

`extract_source_variables`(variables, varname, ...)

Support function to extract the “atomic” variables used in a variable that is of instance `Subexpression`.

---

## extract\_source\_variables function

(Shortest import: `from brian2genn.device import extract_source_variables`)

`brian2genn.device.extract_source_variables` (variables, varname, smvariables)

Support function to extract the “atomic” variables used in a variable that is of instance `Subexpression`.

---

`freeze`(code, ns)

Support function for substituting constant values.

---



## freeze function

(Shortest import: `from brian2genn.device import freeze`)

`brian2genn.device.freeze` (*code, ns*)

Support function for substituting constant values.

---

`get_compile_args()`

Get the compile args based on the users preferences.

---

## get\_compile\_args function

(Shortest import: `from brian2genn.device import get_compile_args`)

`brian2genn.device.get_compile_args` ()

Get the compile args based on the users preferences. Uses Brian's preferences for the C++ compilation (either `codegen.cpp.extra_compile_args` for both Windows and UNIX, or `codegen.cpp.extra_compile_args_gcc` for UNIX and `codegen.cpp.extra_compile_args_msvc` for Windows), and the Brian2GeNN preference `devices.genn.extra_compile_args_nvcc` for the CUDA compilation with nvcc.

**Returns** (`compile_args_gcc, compile_args_msvc, compile_args_nvcc`) : (str, str, str)

Tuple with the respective compiler arguments (as strings).

---

`get_genn_prefs()`

Get the GeNN preferences that are exposed in brian2genn user preferences.

---

## get\_genn\_prefs function

(Shortest import: `from brian2genn.device import get_genn_prefs`)

`brian2genn.device.get_genn_prefs` ()

Get the GeNN preferences that are exposed in brian2genn user preferences. Uses the Brian2GeNN preferences `devices.genn.auto_choose_device` and `devices.genn.default_device`

**Returns** (`genn_auto_choose_device, genn_default_device`) : (int, int)

Tuple with the genn preference settings.

---

`stringify`(code)

Helper function to prepare multiline strings (potentially including quotation marks) to be included in strings.

---

## stringify function

(Shortest import: `from brian2genn.device import stringify`)

`brian2genn.device.stringify` (*code*)

Helper function to prepare multiline strings (potentially including quotation marks) to be included in strings.

**Parameters** `code` : str

The code to convert.

## Objects

<code>genn_device</code>	The main “genn” device.
--------------------------	-------------------------

## genn\_device object

(Shortest import: `from brian2genn.device import genn_device`)

`brian2genn.device.genn_device = <brian2genn.device.GeNNDevice object>`

The main “genn” device. This does most of the translation work from Brian 2 generated code to functional GeNN code, assisted by the “GeNN language”.

## genn\_generator module

The code generator for the “genn” language. This is mostly C++ with some specific decorators (mainly “`__host__ __device__`”) to allow operation in a CUDA context.

Exported members: `GeNNCodeGenerator`

### Classes

<code>GeNNCodeGenerator(*args, **kwds)</code>	“GeNN language”
---	-----------------

## GeNNCodeGenerator class

(Shortest import: `from brian2genn.genn_generator import GeNNCodeGenerator`)

**class** `brian2genn.genn_generator.GeNNCodeGenerator(*args, **kwds)`

Bases: `brian2.codegen.generators.base.CodeGenerator`

“GeNN language”

For user-defined functions, there are two keys to provide:

**support\_code** The function definition which will be added to the support code.

**hashdefine\_code** The `#define` code added to the main loop.

### Attributes

<code>flush_denormals</code>
<code>restrict</code>

### Methods

<code>denormals_to_zero_code()</code>
<code>determine_keywords()</code>
<code>get_array_name(var[, access_data])</code>
<code>translate_expression(expr)</code>
<code>translate_one_statement_sequence(statements)</code>
<code>translate_statement(statement)</code>

Continued on next page

Table 5.30 – continued from previous page

---

```
translate_to_declarations(statements)
translate_to_read_arrays(statements)
translate_to_statements(statements)
translate_to_write_arrays(statements)
```

---

**Details**

```
flush_denormals
restrict
denormals_to_zero_code ()
determine_keywords ()
static get_array_name (var, access_data=True)
translate_expression (expr)
translate_one_statement_sequence (statements, scalar=False)
translate_statement (statement)
translate_to_declarations (statements)
translate_to_read_arrays (statements)
translate_to_statements (statements)
translate_to_write_arrays (statements)
```

**Functions**


---

<code><i>get_var_ndim</i>(<i>v</i>[, <i>default_value</i>])</code>	Helper function to get the <code>ndim</code> attribute of a <code>DynamicArrayVariable</code> , falling back to the previous name dimensions if necessary.
--	--

---

**get\_var\_ndim function**

(Shortest import: `from brian2genn.genn_generator import get_var_ndim`)

`brian2genn.genn_generator.get_var_ndim(v, default_value=None)`

Helper function to get the `ndim` attribute of a `DynamicArrayVariable`, falling back to the previous name dimensions if necessary.

**Parameters** *v* : `ArrayVariable`

The variable for which to retrieve the number of dimensions.

**default\_value** : optional

A default value if the attribute does not exist

**Returns** `ndim` : int

Number of dimensions

## insyn module

GeNN accumulates postsynaptic changes into a variable `inSyn`. The idea of this module is to check, for a given Synapses, whether or not it can be recast into this formulation, and if so to relabel the variables appropriately.

In GeNN, each synapses object has an associated variable `inSyn`. The idea is that we will do something like this in Brian terms:

```
v += w (synapses code) dv/dt = -v/tau (neuron code)
```

should be replaced by:

```
inSyn += w (synapses code) dv/dt = -v/tau (neuron code) v += inSyn; inSyn = 0; (custom operation carried out after integration step)
```

The reason behind this organisation in GeNN is that the communication of spike events and the corresponding updates of post-synaptic variables are separated out for better performance. In principle all kinds of operations on the pre- and post-synaptic variables can be allowed but with a heavy hit in the computational speed.

The conditions for this rewrite to be possible are as follows for presynaptic event code: - Each expression is allowed to modify synaptic variables. - An expression can modify a neuron variable only in the following ways:

```
neuron_var += expr (where expr contains only synaptic variables) neuron_var = expr (where expr- neuron_var can be simplified to contain only synaptic variables)
```

- The set of modified neuron variables can only have one element

And for the postsynaptic code, only synaptic variables can be modified.

The output of this code should be: - Raise an error if it is not possible, explaining why - Replace the line `neuron_var (+)= expr` with `addtoinSyn = new_expr` - Return `neuron_var` so that it can be used appropriately in `GeNNDevice.build`

The GeNN syntax is:

```
addtoinSyn = expr
```

Brian codegen implementation:

I think the correct place to start is given a Statement sequence for a Synapses pre or post code object, check the conditions. Then, we need to create two additional CodeObjects which overwrite `translate_one_statement_sequence` to call this function and rewrite the appropriate statement.

### Functions

---

```
check_pre_code(codegen, stmts, vars_pre, ...)
```

Given a set of statements `stmts` where the variables names in `vars_pre` are presynaptic, in `vars_syn` are synaptic and in `vars_post` are postsynaptic, check that the conditions for compatibility with GeNN are met, and return a new statement sequence translated for compatibility with GeNN, along with the name of the targeted variable.

---

### check\_pre\_code function

(Shortest import: `from brian2genn.insyn import check_pre_code`)

```
brian2genn.insyn.check_pre_code(codegen, stmts, vars_pre, vars_syn, vars_post, conditional_write_vars)
```

Given a set of statements `stmts` where the variables names in `vars_pre` are presynaptic, in `vars_syn` are synaptic and in `vars_post` are postsynaptic, check that the conditions for compatibility with GeNN are met, and return a new statement sequence translated for compatibility with GeNN, along with the name of the targeted variable.

Also adapts the synaptic statement to be multiplied by 0 for a refractory post-synaptic cell.

## preferences module

Preferences that relate to the brian2genn interface.

## Subpackages



## CHAPTER 6

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`





**b**

`brian2genn.binomial`, 13

**c**

`brian2genn.codeobject`, 13

`brian2genn.correctness_testing`, 14

**d**

`brian2genn.device`, 15

**g**

`brian2genn.genn_generator`, 22

**i**

`brian2genn.insyn`, 24

**p**

`brian2genn.preferences`, 25



**A**

activate() (brian2genn.device.GeNNDevice method), 17  
 add\_array\_variable() (brian2genn.device.GeNNDevice method), 17  
 add\_array\_variables() (brian2genn.device.GeNNDevice method), 17  
 add\_parameter() (brian2genn.device.GeNNDevice method), 18

**B**

before\_run() (brian2genn.correctness\_testing.GeNNConfiguration method), 14  
 before\_run() (brian2genn.correctness\_testing.GeNNConfiguration method), 15  
 before\_run() (brian2genn.correctness\_testing.GeNNConfiguration method), 15  
 brian2genn.binomial (module), 13  
 brian2genn.codeobject (module), 13  
 brian2genn.correctness\_testing (module), 14  
 brian2genn.device (module), 15  
 brian2genn.genn\_generator (module), 22  
 brian2genn.insyn (module), 24  
 brian2genn.preferences (module), 25  
 build() (brian2genn.device.GeNNDevice method), 18

**C**

check\_pre\_code() (in module brian2genn.insyn), 24  
 code\_object() (brian2genn.device.GeNNDevice method), 18  
 code\_object\_class() (brian2genn.device.GeNNDevice method), 18  
 collect\_synapses\_variables() (brian2genn.device.GeNNDevice method), 18  
 compile\_source() (brian2genn.device.GeNNDevice method), 18  
 copy\_source\_files() (brian2genn.device.GeNNDevice method), 18  
 CPPWriter (class in brian2genn.device), 15

**D**

decorate() (in module brian2genn.device), 20  
 DelayedCodeObject (class in brian2genn.device), 16  
 denormals\_to\_zero\_code() (brian2genn.genn\_generator.GeNNCodeGenerator method), 23  
 determine\_keywords() (brian2genn.genn\_generator.GeNNCodeGenerator method), 23

**E**

extract\_source\_variables() (in module brian2genn.device), 20  
 CPU  
 fill\_with\_array() (brian2genn.device.GeNNDevice method), 18  
 fix\_random\_generators() (brian2genn.device.GeNNDevice method), 18  
 fix\_synapses\_code() (brian2genn.device.GeNNDevice method), 18  
 flush\_denormals (brian2genn.genn\_generator.GeNNCodeGenerator attribute), 23  
 freeze() (in module brian2genn.device), 21

**G**

generate\_code\_objects() (brian2genn.device.GeNNDevice method), 18  
 generate\_engine\_source() (brian2genn.device.GeNNDevice method), 18  
 generate\_main\_source() (brian2genn.device.GeNNDevice method), 18  
 generate\_makefile() (brian2genn.device.GeNNDevice method), 18  
 generate\_model\_source() (brian2genn.device.GeNNDevice method), 18  
 generate\_objects\_source() (brian2genn.device.GeNNDevice method), 18

genn\_device (in module brian2genn.device), 22  
 GeNNCodeGenerator (class in brian2genn.genn\_generator), 22  
 GeNNCodeObject (class in brian2genn.codeobject), 13  
 GeNNConfiguration (class in brian2genn.correctness\_testing), 14  
 GeNNConfigurationCPU (class in brian2genn.correctness\_testing), 14  
 GeNNConfigurationOptimized (class in brian2genn.correctness\_testing), 15  
 GeNNDevice (class in brian2genn.device), 16  
 GeNNUserCodeObject (class in brian2genn.codeobject), 14

get\_array\_name() (brian2genn.genn\_generator.GeNNCodeGenerator static method), 23  
 get\_compile\_args() (in module brian2genn.device), 21  
 get\_genn\_prefs() (in module brian2genn.device), 21  
 get\_var\_ndim() (in module brian2genn.genn\_generator), 23

## M

make\_main\_lines() (brian2genn.device.GeNNDevice method), 18

## N

network\_run() (brian2genn.device.GeNNDevice method), 18  
 neuronModel (class in brian2genn.device), 19

## P

process\_neuron\_groups() (brian2genn.device.GeNNDevice method), 18  
 process\_poisson\_groups() (brian2genn.device.GeNNDevice method), 18  
 process\_rate\_monitors() (brian2genn.device.GeNNDevice method), 18  
 process\_spike\_monitors() (brian2genn.device.GeNNDevice method), 18  
 process\_spikegenerators() (brian2genn.device.GeNNDevice method), 18  
 process\_state\_monitors() (brian2genn.device.GeNNDevice method), 18  
 process\_synapses() (brian2genn.device.GeNNDevice method), 19

## R

rateMonitorModel (class in brian2genn.device), 19  
 restrict (brian2genn.genn\_generator.GeNNCodeGenerator attribute), 23

run() (brian2genn.device.GeNNDevice method), 19

## S

source\_files (brian2genn.device.GeNNDevice attribute), 17  
 spikegeneratorModel (class in brian2genn.device), 19  
 spikeMonitorModel (class in brian2genn.device), 19  
 stateMonitorModel (class in brian2genn.device), 20  
 stringify() (in module brian2genn.device), 21  
 synapseModel (class in brian2genn.device), 20

## T

translate\_expression() (brian2genn.genn\_generator.GeNNCodeGenerator method), 23  
 translate\_one\_statement\_sequence() (brian2genn.genn\_generator.GeNNCodeGenerator method), 23  
 translate\_statement() (brian2genn.genn\_generator.GeNNCodeGenerator method), 23  
 translate\_to\_declarations() (brian2genn.genn\_generator.GeNNCodeGenerator method), 23  
 translate\_to\_read\_arrays() (brian2genn.genn\_generator.GeNNCodeGenerator method), 23  
 translate\_to\_statements() (brian2genn.genn\_generator.GeNNCodeGenerator method), 23  
 translate\_to\_write\_arrays() (brian2genn.genn\_generator.GeNNCodeGenerator method), 23

## V

variableview\_set\_with\_expression() (brian2genn.device.GeNNDevice method), 19  
 variableview\_set\_with\_expression\_conditional() (brian2genn.device.GeNNDevice method), 19  
 variableview\_set\_with\_index\_array() (brian2genn.device.GeNNDevice method), 19

## W

write() (brian2genn.device.CPPWriter method), 16