Brian2GeNN Documentation

Release 1.5

Brian2GeNN authors

Apr 29, 2020

Contents

Contents:

CHAPTER 1

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.

1.1 Installing the Brian2GeNN interface

In order to use the Brian2GeNN interface, all three Brian 2, GeNN and Brian2GeNN need to be fully installed. To install GeNN and Brian 2, refer to their respective documentation:

- Brian 2 installation instructions
- GeNN installation instructions

Note that you will have to also 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_backend.cuda_path* preference) to point to CUDA's installation directory.

To install brian2genn, use `pip`:

```
pip install brian2genn
```

(might require administrator privileges depending on the configuration of your system; add `--user` to force an installation with user privileges only).

As detailed in the GeNN installation instructions), you also need to ensure that GeNN's bin directory is added to your path. Alternatively, you can set the *devices.genn.path* preference to your GeNN directory to achieve the same effect.

Note: We no longer provide conda packages for Brian2GeNN. Conda packages for previous versions of Brian2GeNN have been tagged with the `archive` label and are still available in the `brian-team` channel.

1.2 Using the Brian2GeNN interface

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

```
import brian2genn
```

The 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*.

CHAPTER 2

Unsupported features in Brian2GeNN

2.1 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.

2.2 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',
```

(continues on next page)

(continued from previous page)

```
                        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.

2.3 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.

2.4 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.

2.5 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.

2.6 Timed arrays

Timed arrays post 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 inefficiences, the use of `TimedArray` is therefore currently restricted to code in `run_regularly` operations that will be executed on the CPU.

2.7 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.

2.8 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 `device.reinit()` and `device.activate()` 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.

2.9 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).

2.10 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.

CHAPTER 3

Brian2GeNN specific preferences

3.1 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'
```

3.2 Compiler preferences

On Linux and Mac, Brian2GeNN will use the compiler preferences specified for Brian2 for compiling the executable. This means you should set the `codegen.cpp.extra_compile_args` preference, or `codegen.cpp.extra_compile_args_gcc`

Brian2GeNN also offers a preference to specify additional compiler flags for CUDA compilation on Linux and Mac with the nvcc compiler: *devices.genn.cuda_backend.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.cuda_backend.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.

3.3 CUDA preferences

The `devices.genn.cuda_backend` preferences contain CUDA-specific preferences. If you have multiple CUDA devices you can manually select a device like this:

```
prefs.devices.genn.cuda_backend.device_select = 'MANUAL'
prefs.devices.genn.cuda_backend.manual_device = 1
```

Normally GeNN automatically optimizes the 'block size' used for its CUDA kernels but this can also be overriden like:

```
prefs.devices.genn.cuda_backend.blocksize_select_method = 'MANUAL'
prefs.devices.genn.cuda_backend.neuron_blocksize = 64
```

`pre_neuron_reset_blocksize`, `pre_synapse_reset_blocksize`, `synapse_blocksize`, `learning_blocksize`, `synapse_dynamics_blocksize`, `init_blocksize` and `init_sparse_blocksize` can also be configured in this way.

3.4 List of preferences

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.kernel_timing = False This preference determines whether GeNN should record kernel runtimes; note that this can affect performance.

devices.genn.path = None The path to the GeNN installation (if not set, the version of GeNN in the path will be used instead)

devices.genn.synapse_span_type = 'POSTSYNAPTIC' This preference determines whether the spanType (parallelization mode) for a synapse population should be set to pre-synapstic or post-synaptic.

Preferences that relate to the CUDA backend for the brian2genn interface

devices.genn.cuda_backend.blocksize_select_method = 'OCCUPANCY' The GeNN preference blockSizeSelectMethod that determines whether GeNN should use its internal algorithms to optimise the different block sizes.

devices.genn.cuda_backend.cuda_path = None The path to the CUDA installation (if not set, the CUDA_PATH environment variable will be used instead)

devices.genn.cuda_backend.device_select = 'OPTIMAL' The GeNN preference deviceSelectMethod that determines how to chose which GPU device to use.

devices.genn.cuda_backend.extra_compile_args_nvcc = [] Extra compile arguments (a list of strings) to pass to the nvcc compiler.

devices.genn.cuda_backend.init_blocksize = 32 The GeNN preference initBlockSize that determines the CUDA block size for the neuron kernel if blocksize_select_method is set to MANUAL.

devices.genn.cuda_backend.init_sparse_blocksize = 32 The GeNN preference initSparseBlockSize that determines the CUDA block size for the neuron kernel if blocksize_select_method is set to MANUAL.

devices.genn.cuda_backend.learning_blocksize = 32 The GeNN preference learningBlockSize that determines the CUDA block size for the neuron kernel if blocksize_select_method is set to MANUAL.

devices.genn.cuda_backend.manual_device = 0 The GeNN preference manualDeviceID that determines CUDA enabled device should be used if device_select is set to MANUAL.

devices.genn.cuda_backend.neuron_blocksize = 32 The GeNN preference neuronBlockSize that determines the CUDA block size for the neuron kernel if blocksize_select_method is set to MANUAL.

devices.genn.cuda_backend.pre_neuron_reset_blocksize = 32 The GeNN preference preNeuronResetBlockSize that determines the CUDA block size for the pre-neuron reset kernel if blocksize_select_method is set to MANUAL.

devices.genn.cuda_backend.pre_synapse_reset_blocksize = 32 The GeNN preference preSynapseResetBlockSize that determines the CUDA block size for the pre-synapse reset kernel if blocksize_select_method is set to MANUAL.

devices.genn.cuda_backend.synapse_blocksize = 32 The GeNN preference synapseBlockSize that determines the CUDA block size for the neuron kernel if blocksize_select_method is set to MANUAL.

devices.genn.cuda_backend.synapse_dynamics_blocksize = 32 The GeNN preference synapseDynamicsBlockSize that determines the CUDA block size for the neuron kernel if blocksize_select_method is set to MANUAL.

CHAPTER 4

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.

4.1 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 mdoel 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 capsulated into `GeNNCodeObjects`.

4.2 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.

4.3 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.

4.4 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 transfered 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.

4.5 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.

CHAPTER 5

brian2genn package

x.__init__(. . .) initializes x; see help(type(x)) for signature

5.1 `binomial` module

Implementation of `BinomialFunction`

5.2 `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

`GeNNCodeObject`(owner, code, variables, . . .)	Class of code objects that generate GeNN "code snippets"

5.2.1 GeNNCodeObject class

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

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

Bases: `brian2.codegen.codeobject.CodeObject`

Class of code objects that generate GeNN "code snippets"

Methods

after_run()	
before_run()	
compile()	
run()	Runs the code in the namespace.

Details

after_run()

before_run()

compile()

run()

Runs the code in the namespace.

Returns **return_value** : dict

A dictionary with the keys corresponding to the `output_variables` defined during the call of `CodeGenerator.code_object`.

GeNNUserCodeObject(owner, code, variables, ...)	Class of code objects that generate GeNN "user code"

5.2.2 GeNNUserCodeObject class

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

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

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

Class of code objects that generate GeNN "user code"

5.3 `correctness_testing` module

Definitions of the configuration for correctness testing.

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

Classes

GeNNConfiguration([maximum_run_time])	

Methods

5.3.1 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

5.3.2 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

5.3.3 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()

5.4 `device` module

Module implementing the bulk of the brian2genn interface by defining the "genn" device.

Exported members: `GeNNDevice`

Classes

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

5.4.1 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).

5.4.2 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).

Methods

after_run()	
before_run()	

Details

after_run()

before_run()

GeNNDevice()	The main "genn" device.

5.4.3 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])	Called when this device is set as the current device.
add_array_variable(model, varname, variable)	
add_array_variables(model, owner)	
add_parameter(model, varname, variable)	
build([directory, compile, run, use_GPU, . . .])	This function does the main post-translation work for the genn device.
code_object(owner, name, abstract_code, . . .)	Processes abstract code into code objects and stores them in different arrays for `GeNNCodeObjects` and `GeNNUserCodeObjects`.

Continued on next page

Table 16 – continued from previous page

code_object_class([codeobj_class])	Return `CodeObject` class (either `CPPStandaloneCodeObject` class or input)
collect_synapses_variables(synapse_model, ...)	
compile_source(debug, directory, use_GPU)	
copy_source_files(writer, directory)	
fill_with_array(var, arr)	Fill an array with the values given in another array.
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, ...)	
generate_code_objects(writer)	
generate_engine_source(writer, objects)	
generate_main_source(writer, main_lines)	
generate_makefile(directory, use_GPU)	
generate_max_row_length_code_objects(writer)	
generate_model_source(writer, main_lines, ...)	
generate_objects_source(arange_arrays, net, ...)	
insert_code(slot, code)	Insert custom C++ code directly into `main.cpp`.
make_main_lines()	Generates the code lines that handle initialisation of Brian 2 cpp_standalone type arrays.
network_run(net, duration[, report, ...])	
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, ...)	
process_synapses(synapse_groups, objects)	
run(directory, use_GPU, with_output)	
variableview_set_with_expression(...[, ...])	
variableview_set_with_expression_conditional(...)	
variableview_set_with_index_array(...)	

Details

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

activate(*build_on_run=True*, ***kwargs*)
Called when this device is set as the current device.

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 cod 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*, ***kwds*)

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

code_object_class(*codeobj_class=None*, **args*, ***kwds*)

Return `CodeObject` class (either `CPPStandaloneCodeObject` class or input)

Parameters **codeobj_class** : a `CodeObject` class, optional

If this is keyword is set to None or no arguments are given, this method will return the default (`CPPStandaloneCodeObject` class).

fallback_pref : str, optional

For the cpp_standalone device this option is ignored.

Returns **codeobj_class** : class

The `CodeObject` class that should be used

collect_synapses_variables(*synapse_model*, *pathway*, *codeobj*)

compile_source(*debug*, *directory*, *use_GPU*)

copy_source_files(*writer*, *directory*)

fill_with_array(*var*, *arr*)

Fill an array with the values given in another array.

Parameters **var** : `ArrayVariable`

The array to fill.

arr : `ndarray`

The array values that should be copied to `var`.

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*, *objects*)

generate_main_source(*writer*, *main_lines*)

generate_makefile(*directory*, *use_GPU*)

generate_max_row_length_code_objects(*writer*)

generate_model_source(*writer*, *main_lines*, *use_GPU*)

generate_objects_source(*arange_arrays*, *net*, *static_array_specs*, *synapses*, *writer*)

insert_code(*slot*, *code*)

Insert custom C++ code directly into `main.cpp`. The available slots are:

before_start / after_start Before/after allocating memory for the arrays and loading arrays from disk.

before_run / after_run Before/after calling GeNN's `run` function.

before_end / after_end Before/after writing results to disk and deallocating memory.

Parameters **slot** : str

The name of the slot where the code will be placed (see above for list of available slots).

code : str

The C++ code that should be inserted.

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*, *objects*)

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*)

neuronModel()	Class that contains all relevant information of a neuron model.

5.4.4 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.

rateMonitorModel()	CLass that contains all relevant information about a rate monitor.

5.4.5 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.

spikeMonitorModel()	Class the contains all relevant information about a spike monitor.

5.4.6 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.

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

5.4.7 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 relvant information about a state monitor.

5.4.8 stateMonitorModel class

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

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

Bases: `object`

Class that contains all relvant information about a state monitor.

synapseModel()	Class that contains all relevant information about a synapse model.

5.4.9 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 $(.).

5.4.10 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`.

5.4.11 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`.

find_executable(executable)	Tries to find 'executable' in the path

5.4.12 find_executable function

(*Shortest import*: `from brian2genn.device import find_executable`)

`brian2genn.device.`**find_executable**(*executable*)

Tries to find 'executable' in the path

Modified version of distutils.spawn.find_executable as this has stupid rules for extensions on Windows. Returns the complete filename or None if not found.

freeze(code, ns)	Support function for substituting constant values.

5.4.13 freeze function

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

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

Support function for substituting constant values.

get_gcc_compile_args()	Get the compile args for GCC based on the users preferences.

5.4.14 get_gcc_compile_args function

(*Shortest import*: `from brian2genn.device import get_gcc_compile_args`)

`brian2genn.device.`**`get_gcc_compile_args`**`()`

Get the compile args for GCC based on the users preferences. Uses Brian's preferences for the C++ compilation (either `codegen.cpp.extra_compile_args` or `codegen.cpp.extra_compile_args_gcc`).

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

Tuple with the respective compiler arguments (as strings).

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

5.4.15 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

genn_device	The main "genn" device.

5.4.16 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".

5.5 `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

GeNNCodeGenerator(*args, **kwds)	"GeNN language"

5.5.1 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

flush_denormals
restrict

Methods

denormals_to_zero_code()	
determine_keywords()	A dictionary of values that is made available to the templated.
get_array_name(var[, access_data])	
translate_expression(expr)	Translate the given expression string into a string in the target language, returns a string.
translate_one_statement_sequence(statements)	
translate_statement(statement)	Translate a single line `Statement` into the target language, returns a string.
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()

A dictionary of values that is made available to the templated. This is used for example by the `CPPCodeGenerator` to set up all the supporting code

static get_array_name(*var*, *access_data=True*)

translate_expression(*expr*)
Translate the given expression string into a string in the target language, returns a string.

translate_one_statement_sequence(*statements*, *scalar=False*)

translate_statement(*statement*)
Translate a single line `Statement` into the target language, returns a string.

translate_to_declarations(*statements*)

translate_to_read_arrays(*statements*)

translate_to_statements(*statements*)

translate_to_write_arrays(*statements*)

Functions

get_var_ndim(v[, default_value])	Helper function to get the `ndim` attribute of a `DynamicArrayVariable`, falling back to the previous name `dimensions` if necessary.

5.5.2 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

5.6 `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 priniciple 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.

5.6.1 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.

5.7 `preferences` module

Preferences that relate to the brian2genn interface.

Classes

DeprecatedValidator(message)	'Validator' for deprecated preferences

5.7.1 DeprecatedValidator class

(*Shortest import*: `from brian2genn.preferences import DeprecatedValidator`)

class brian2genn.preferences.**DeprecatedValidator**(*message*)

Bases: `object`

'Validator' for deprecated preferences

Used as a validator for preferences that have been (rudely) deprecated

Methods

`__call__`(...)	<==> x(...)

Details

__call__(*...*) <==> *x(...)*

5.8 Subpackages

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

_

b

c

d

g

i

p

Index

Symbols

A

B

C

D

E

T

V

W