cmpneuron Documentation

Jonny Quarta

May 9, 2013

1 Introduction

The cmpneuron tool has been created in order to allow users to easily write their custom neuron models in python, without any knowledge of C++.
The first important thing to know is that for correct behaviour, at least Cython 0.18 must be installed.

You can invoke the tool by typing
cmpneuron [**filename**]
or
cmpneuron [**option**]
where *filename* means the python file containing the neuron class and *option* can be

- –help : Prints a little help
- –doc : Opens this documentation

There are some important rules to take into account when compiling a new neuron:

- The terminal must be situated in the same folder as the python file (or files).
- *filename* must be written without the extension (ex: myneuron and NOT myneuron.py)

2 Writing a new neuron

This section deals with the basic operations one has to perform in order to write a custom neuron. The neuron class has to follow a certain structure, which will be explained during the construction of a little example.
Let assume we want to create a new custom neuron, which we will call **test_neuron**. Then we create a new file, *test_neuron.py*. **Note that the file name and the neuron name has to be the same** (in this case test_neuron). Once the file is created, we can put our skeleton code:

```
class test_neuron(Neuron):
    def __init__(self):
        pass

    def __dealloc__(self):
        pass

    def calibrate(self):
        pass

    def setStatus(self):
        pass

    def getStatus(self):
        pass

    def update(self):
        pass
```

As we can see, the test_neuron class extends from Neuron. **This is mandatory**. In addition, some methods can be overwritten (these are situated in the base class Neuron). Method overwriting is optional, however at least calibrate and update should be changed, otherwise the neuron will be useless.

Here's a brief description of every method :

- __init__ : this is the constructor. Here's should be declared and initialized the totality of internal members the neuron will use.
- __dealloc__ : this is the destructor.
- getStatus : this method is called whenever the user calls the CyNEST function **cynest.GetStatus(...)** in the python terminal. It is used in very special cases, hence optional.
- setStatus : this method is called whenever the user calls the CyNEST function **cynest.SetStatus(...)** or **cynest.SetDefaults(...)** in the python terminal. This could be usefull if the neuron has to adapt some other members with respect to special ones.
- calibrate : this method is called before the simulation start. Its purpose is to adjust the various neuron parameters before the simulation.
- update : this is the most important method and is called repeatedly during a simulation. Its purpose is to update the various neuron parameters according to the inputs (everything will be explained later).

One very good point is that python can be used in almost every fashion, however a very important fact has to be understood: cmpneuron is based on cython,

therefore every cython limitation translates to a cmpneuron limitation.
Aside from that, one could even write cython code, which would be correctly compiled. The only crucial thing is to keep the base neuron class as previously explained. Of course, custom members can be created (with some limitations, which will be explained later), other python scripts can be imported and other classes can be written in the same base file.

3 Restrictions

As everything on Earth, nothing is perfect and few restrictions have to be followed in order to write compilable code. Here's the list:

- Some special neuron members are already defined and their type cannot be changed and their values should not be updated (not always true, see next section).
- If the neuron declares a new member of custom type (maybe defined in another file), its name must be written with an underscore at the beginning (ex: _myRectangle and NOT myRectangle, for an object of Rectangle type). This "makes" the parameter private and CyNEST will not try to pass it to the python terminal (custom objects cannot be translated into C++ objects). However, if the member has a builtin type (integer, boolean, double, dictionary, list, etc...), there is no problem.
- Every other cython limitation also applies.

4 Standard Parameters and Special Functions

This section deals with the communication between the custom neuron and other ones.

Standard Parameters

As one could imagine, not everything can be done when writing a custom neuron, compared to writing a static C++ compiled with CyNEST neuron. In particular, events cannot be accessed (SpikeEvent, CurrentEvent, etc...).
However, these events are handled in a very general way, providing almost the same confort as normal neuron do. In particular, some standard parameters have been declared and are updated before every call to to *update* method. Here's the list:

- ex_spikes : this is the number of excitatory spikes incoming at a certain point. Note that this is a double value, because every spike in automatically multiplied by it's weight, which couldn't be an integer.

- in_spikes : this is the number of inhibitory spikes incoming at a certain point. Note that this is a double value, because every spike in automatically multiplied by it's weight, which couldn't be an integer. Also note that the value is always negative, because the weight are. If one want to sum all the spikes, he should write *ex_spikes + in_spikes* and not *ex_spikes - in_spikes*.
- currents : this is the currents upcoming at a certain point in time. This is also a double value.
- t_lag : this integer value represents the current point in time (this would be the t variable of a function).
- spike : this is the only modifiable standard parameter. That's the neuron output and is an integer value (not a boolean one). Set it at 1 if the neuron spikes and 0 otherwise.

Special Functions

When writing a neuron, one could need some other special parameters concerning the way the simulation has been set. In a normal C++ neuron, the totality of thse values would be accessible from the Time and Scheduler classes (for more information about these classes, see the NEST official documentation), but these classes are not visible from the custom neuron side. There is however a way in order to acces the most important values and the base Neuron class provides an entire list of methods (15 in total):

- get_ms_on_resolution()
- get_ms_on_tics(tics)
- get_ms_on_steps(steps)
- get_tics_on_resolution(self)
- get_tics_on_steps(steps)
- get_tics_on_ms(ms)
- get_tics_on_ms_stamp(ms_stamp)
- get_steps_on_resolution()
- get_steps_on_tics(tics)
- get_steps_on_ms(ms)
- get_steps_on_ms_stamp(ms_stamp)
- get_modulo(value)
- get_slice_modulo(value)

- get_min_delay()
- get_max_delay()

As an example, writing *self.get_ms_on_resolution()* is the same as writing (in the NEST code, C++) *Time::get_resolution().get_ms()* and *self.get_steps_on_ms(ms)* the same as *Time(Time::ms(ms)).get_steps()*. In addition, writing *self.get_min_delay()* could be compared to the C++ *Scheduler::get_min_delay()*.
Note that calling these functions takes much time because the call spreads through many functions in order to achieve the Time or Scheduler classes. It is therefore advised to avoid these calls in the update method and to create variables instead.
Also note that these special functions cannot be overwritten.