Writing Customable Cython Neurons
User Documentation

Jonny Quarta

August 19, 2013

1 Introduction

CyNEST proposes the possibility to create custom neuron models in cython, without any knowledge of C++. The first important thing to know is that for correct behaviour, at least Cython 0.18 must be installed.
For any information about Cython, from the installation to the language syntax, please consult the home page (www.cython.org).

The main strength of this new feature is the conjunction of use easiness with very little performance loss.

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.pyx*. **Note that the file name and the neuron name have to be the same**, (in this case test_neuron). Once the file is created, we can put our skeleton code:

```
include "[installation path]/include/Neuron.pyx"

cdef class test_neuron(Neuron):
    cdef ....

    def __cinit__(self):
        pass

    cpdef calibrate(self):
        pass
```

```
cpdef setStatus(self, params):
    pass

cpdef getStatus(self):
    pass

cpdef update(self):
    pass
```

As we can see, the test_neuron class extends from Neuron. **This is mandatory**. Also note that the Neuron base class is contained in a special file located at *[installation path]/include/Neuron.pyx*, where the installation path is the path where CyNEST has been installed.
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 :

- __cinit__ : this is the constructor. Here should be initialized the totality of internal members the neuron will use. Note that since this is a cython class, one cannot just declare a new member, as these new objects are not dinamically attached to the class. In order to declare a new member, one has to make a *cdef ...* outside the methods.
- getStatus : this method is called whenever the user calls the CyNEST function **cynest.GetStatus(...)** in the python terminal (or NEST internally calls it). Note that it should output a dictionary containing the totality of the members the user created.
- setStatus : this method is called whenever the user calls the CyNEST function **cynest.SetStatus(...)** or **cynest.SetDefaults(...)** in the python terminal (or NEST internally calls it). This could be usefull if the neuron has to adapt some other members with respect to special ones. Note that it takes as argument a dictionary containing the totality of the updated paramaters the user created. The user should update the model with them.
- 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).

Note that the way the user handles the model parameters is completely unimportant to NEST, as long as they are correctly interfaced with the system (using the *setStatus* and *getStatus* methods). Here is a standard way of doing this:

```
cpdef getStatus(self):
    cdef dict d = {}

    d["p1"] = self.p1
    d["p2"] = self.p2
    ...
    return d

cpdef setStatus(self, d):
    if d.has_key("p1"):
        self.p1 = d["p1"]
    if d.has_key("p2"):
        self.p2 = d["p2"]
    ...
```

Note that the *setStatus* tests are important because one never knows which elements the system wants to set (the others are not present).

3 Standard Parameters and Special Objects

This section deals with the communication between the custom neuron and other ones.
As one could imagine, not everything can be done when writing a custom neuron with respect to writing a static C++ one. As an example, events cannot be accessed (SpikeEvent, CurrentEvent, etc...). However, these situations are handled in a very general way, providing almost the same confort as normal neurons do.

3.1 Standard Parameters

During a model simulation, the communication between the custom neuron and the kernel is very important. In particular, the kernel should know, at least when he needs, the value of the different neuron members. However, during the actual simulation only a few members are important to NEST and these are called Standard Parameters. These have been declared and are updated before every call to the *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 could be a real value.
- 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 could also be a real value. 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 simulation 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.

All these Standard Parameters are contained in the base Neuron class and, since their behaviour is highly optimized, provide a very little performance loss compared to a native neuron. They do not need to be included in the *setStatus* and *getStatus* methods.

3.2 Special Objects

In a normal neuron model (written inside NEST) there is the possibility to access the Time and Scheduler classes, which contain important parameters about the current simulation. These classes are available on the Python side as well, using an emulation procedure. A Time object can be created as one would do in C++:

```
Time(ms(value))
```

Note that normally the Time class has four nested classes for handling different units: *tic*, *step*, *ms* and *ms_stamp*. These are also available on the Python side (see example just above).
In addition, the Time object contains the following methods:

```
def get_tics(self)

def get_steps(self)

def get_ms(self)

def set_to_zero(self)

def calibrate(self)

def advance(self)

def is_grid_time(self)

def is_neg_inf(self)

def is_pos_inf(self)
```

```
    def is_finite(self)

    def is_step(self)

    def succ(self)

    def pred(self)
```

Static methods are also available:

```
Time_get_resolution()

Time_set_resolution(d)

Time_reset_resolution()

Time_resolution_is_default()

Time_get_ms_per_tic()

Time_get_tics_per_ms()

Time_get_tics_per_step()

Time_get_old_tics_per_step()

Time_get_tics_per_step_default()

Time_min()

Time_max()

Time_pos_inf()

Time_neg_inf()
```

Note that these are not contained in the Time object, but rather directly called (the Time_ specifies that originally the static method comes from the Time object).

In addition to the Time class, the Scheduler has be emulated as well, but contains only four static methods:

```
Scheduler_get_modulo(v)

Scheduler_get_slice_modulo(v)

Scheduler_get_min_delay()

Scheduler_get_max_delay()
```

As an example, writing *Time_get_resolution().get_ms()* is the same as writing (in the NEST code, C++) *Time::get_resolution().get_ms()* and *Time(ms(v)).get_steps()* the same as *Time(Time::ms(v)).get_steps()*. In addition, writing *Scheduler_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 other methods 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.

4 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, 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), it should not be put in the dictionary created during the *getStatus* method. This "makes" the parameter private and CyNEST will not try to convert it from Python to C++ (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.

5 Optimization and Multithreading

Perhaps the major limitation (not discussed yet) of the feature is multithreading. In fact, when trying to increase the number of threads in order to speed up the simulation, one actually encounters a decrease in speed.
This is due to the GIL, which prevents different threads from accessing the same ressources.
This problem is solved by specifying to each neuron a special parameter. After

a neuron instance has been created, it is possible to set the *optimized* parameter to True, like this:

```
cynest.RegisterNeuron("myNeuron")

n = cynest.Create("myNeuron")

cynest.SetStatus(n, [{"optimized":True}])
```

This option prevents the GIL from being used, further increases the speed and, most importantly, makes multithreading available by not slowing too much.
However there is a drawback: the model cannot contain any python object, not even a simple dictionary. Therefore, the user can only write classic C parameters (if this is not respected, segmentation faults will occur). This is why the two options are available.
Note however that this does not prevent one creating a temporary dictionary during the *getStatus* and *setStatus* methods. The standard interfacing protocol is therefore maintained.

6 Using the new neuron

After the custom neuron has been written, it can be simply imported and used in the python terminal. In order to achieve that, the user has to use the *cynest.RegisterNeuron(...)* method and then use it like a normal one. The .pyx file should be placed in the same location as the python terminal has been invoked and no file extension has to be written (just like a normal python import).
Note that when registering the neuron, the pyx file is automatically compiled and linked, so compilation errors could occur. In this case, just check them and fix the neuron model.

7 Performances

The system has been tested during a single-threaded simulation of 40 ms concerning over 1000 randomly connected neurons and compared to their native and SLI counterparts. The results are:

- Native neuron: realtime factor is 0.3254
- SLI neuron: realtime factor is 0.0046
- Cython neuron: realtime factor is 0.2711

We can therefore conclude that the cython neuron is only about 20% slower than the native version, whereas the SLI neuron is up to 70 times slower. Note that the highly optimized version has been tested, the other one being 70% slower.
Note that during different test trials, values could slightly change because of the

random factor introduced in the network.