Writing Customable Cython Neurons
Developer Documentation

Jonny Quarta

August 19, 2013

1 Introduction

The NEST Simulator, which stands for Neural Simulation Tool, has been created with the purpose of enabling complex simulations of point-like neurons. There's a wide range of possible models, from the simple integrate and fire model, to the more complex Hodgkin-Huxley one.
These neurons can be connected throught weighted connexions and their activity recorded using virtual tools (multimeter, spike detector, etc...).

In the NEST code, every model extends from a parent class, implementing only basic functions, such as collecting the incoming spikes. The real model is written in the child classes, so that, using polymorphism, adding new neurons is accomplished relatively easily.
However, each time one wants to add a new neuron, he has to write it in C++ (a difficult language) and recompile the whole project. This makes the tool not so flexible and slows down developpement time.
That's why the need for a plugin system has arised. Since NEST provides an interface enabling the user to execute the simulator from a Python terminal, the system has to be designed in order to facilitate the python utilization. Therefore, this new feature should enable to easily write neurons and import them like normal python objects, these being automatically recognized and executed by NEST. In addition, since C++ is a difficult language, models should be written in a more scientific language such as Cython.

The purpose of this project is the creation of a plugin system having these features. This is accomplished using a relatively new technology, Cython. This special tool enables one to write Python code and compile it into C++ or writing cython code and interface it with Python.
During the next sections, first the NEST structure will be presented, then a global description of the interface will be depicted. After that, more details are given, beginning with the Python side of the project, followed by the C++ side. Finally, performances are analyzed.

2 NEST Structure

The aim of this section is to enable the reader understanding the global NEST structure, which is essential in order to grasp the details of the new interface, presented in the next sections.
Basically, NEST is composed of three major parts:

- The simulation kernel : this is the core of the system, enabling, among others, simulations, events scheduling, communication of neurons and activity recording. The C++ models are also situated in this part.
- SLI Interface : NEST is a high configurable simulator, therefore a powerfull language has been created in order to configure simulation properties, set experiment parameters, create neurons and connexions between them, ect... SLI is loosely based on PostScript, using a stack to put, manipulate and retrieve objects.
- CyNEST Interface : NEST can be used from a Python terminal, configuring it by calling very high functions, which will in turn call corresponding SLI commands. This interface can be accessed by just typing *import cynest* in a Python terminal.

For more information, see the original documentation
(http://www.nest-initiative.org/index.php/Software:About_NEST).

3 Global Interface Structure

This section deals with the global structure of the new interface between CyNEST and custom neurons.

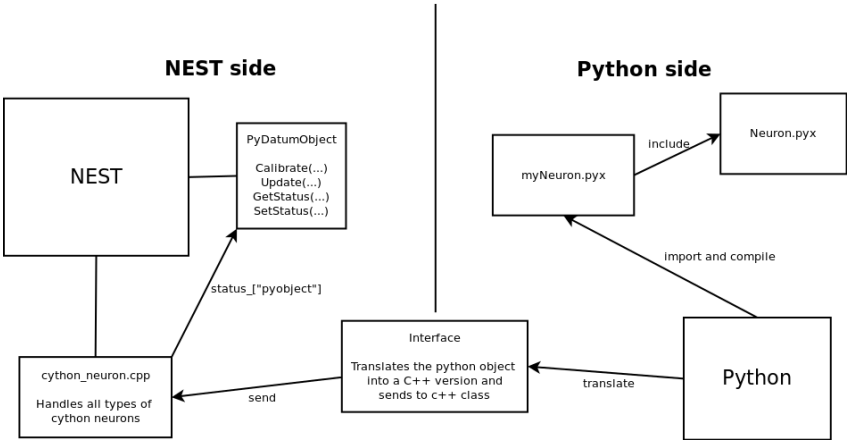

Figure 1: Interface Structure

Figure 1 shows that structure as a diagram. The key idea is quite simple: the user writes his models as .pyx files and imports them into CyNEST via the python terminal. Then, whenever he creates a new instance of a custom neuron, CyNEST will transform it into a C++ version and give it to the *cython_neuron.cpp* class, which will then call the corresponding methods (update, calibrate, etc...). Therefore, the C++ side will use the C/Python API in order to access the python object fields and methods.

During the next sections, the most important regions of the diagram will be covered in much greater detail, so that the reader can have insight into the subtle choises and concepts that populate this innovative feature of NEST.

4 Python Side

This section focuses on the main operations the Python side has to accomplish in order to make the system work.

4.1 Base Neuron Class

First of all, the model has to be written (for details about how to wrtite new models, please consult the user documentation).
However, it is important to know what structure the model relies on. This is the base, called *Neuron* and located in the *"installation folder"/include/Neuron.pyx* file, with the installation path being where CyNEST has been installed. This file contains the following code:

```
cdef class Neuron:
    cdef object time_scheduler
    # Standard Parameters
    cdef double currents
    ...

    def __cinit__(self):
        pass

    cpdef calibrate(self):
        pass

    cpdef update(self):
        pass

    ...


    # We convert the address of the variables into
```

```
    # long in order to extract the pointers

    cpdef getPCurrents(self):
        return <long>(&(self.currents))

    ...
```

Note that not everything is shown.
The class is composed of various parameters, whose usefullness will be discussed later, and other functions, some of which have to be overwritten by the child model. These functions are *calibrate*, *update*, *setStatus* and *getStatus*. The other functions will be explained during the next sections.

4.2 Neuron Registration

Once the model has been completed, the user has to load it before any utilization. This process is called **neuron registration**.
Indeed, the first thing to do is typing *cynest.RegisterNeuron(...)*. This method will use a cython mechanism, called **pyximport**, to dinamically import and compile the .pyx file containing the custom model:

```
def RegisterNeuron(model_name):
    exec("import " + model_name)
    globals()[model_name] = locals()[model_name]
    exec(model_name + ".setScheduler(schedulerObj)")
    exec(model_name + ".setTime(timeObj)")
    ...
    cython_models.append(model_name)
    reg(model_name)
```

Here we can see that the model is compiled and imported. *cython_models* is a list containing the name of the custom models, used when the software has to check if a neuron is normal or custom. The final step is to actually register the model into the system and is accomplished by the *reg* function, which actually corresponds to the *register_cython_model* method contained in the *cynest/cynestkernel.cpp* file. The other instructions contained in exec statements will be discussed in later sections.

When a new model instance is created, NEST will search that particular model among its classes. But how to know if a model has a corresponding class (i.e., if it exists)? The *models/modelsmodule.cpp* class provides the answer. Every new model is registered in the **models dictionary**, so that when a new instance has to be created, NEST just checks in this dictionary, looks for that particular model and creates the corresponding class object.
In order to register a new model, a code such this has to be written :

```
register_model<iaf_cond_alpha>(net_, "iaf_cond_alpha");
```

Here, the first *iaf_cond_alpha* corresponds to the class name, whereas the second one corresponds to the string pointing to that particular class.
Therefore, the *register_cython_model* method, which is actually located in the *cynest/cython_neuron.cpp* file, finally calls a statement like this:

```
register_model<cython_neuron>(net_, model_name);
```

For more information, look to the *models/modelsmodule.cpp* and *cynest/cython_neuron.cpp* files.

4.3 Neuron Creation

After a neuron is registered, new instances are created with the *cynest.Create(..)* method. Here, the main task is to check whether the neuron is a normal or a custom one, as well as sending the model to the c++ class if it is the case:

```
if model in cython_models:
    for i in ids:
        exec("tmpobj___ = " + model + "." + model + "()")
        SetStatus([i], {'pyobject':tmpobj___})
```

The function actually creates an intermediate object corresponding to the custom modeland gives this temporary object to the neuron on the c++ side. Do not forget that for every custom neuron there is, in addition to the python model, also the corresponding c++ class, which always is *cython_neuron.cpp*.

4.4 Python - C++ Transformation

The last step before the *cython_neuron* class can access a functional object is to transform it from Python to C++. Normally, builtin types are already handled by CyNEST, but the need for an additional handling system arises.
Python objects are translated into Datums in the *cynest/cynestkernel.cpp* file, precisely in the *PyObject_as_Datum* method. Here is an example :

```
if (PyFloat_Check(pObj)) // object is float
    return new DoubleDatum(PyFloat_AsDouble(pObj));
```

Using the C/Python API, one can access python objects, check their type and extract their values.
The key idea is therefore to create a new type of Datum, **PyObjectDatum**, which will handle the model (we assume therefore that whenever an object has not a builtin type, it is a neuron model. This is of course not true and checks should be provided. That will be done in another version). Indeed, if no builtin type has not been detected, we simply englobe the python object into a PyObjectDatum one :

```
return new PyObjectDatum(pObj);
```

5 C++ Side

This section deals with the C++ side and the ways the code is optimized for achieving very little performance loss.

5.1 cython_neuron Class

Every custom model is handled by the *cython_neuron* class, which contains the translated object corresponding to a PyObjectDatum object.
Each time one of the custom methods is called, this class will class the apposite method of the PyObjectDatum object, which is contained in the status_ dictionary. Here is an example:

```
if(state_->known(Name("pyobject"))) {
    (*state_)[Name("pyobject")]->
         call_status_method(std::string("setStatus"), &state_);
}
```

The calibrate method is handled more or less the same way:

```
if(state_->known(Name("pyobject"))) {
    pyObj = &(*(*state_)[Name("pyobject")]);

    ...
    (*state_)[Name("pyobject")]->
                     call_method(std::string("calibrate"));
}
```

As one can see, the *pyobject* element is accessed and the calibrate method is called. *pyObj* is extracted and then used during the update for the purpose of efficiency (no element lookup).

5.2 Update Optimization with Standard Parameters

When a neuron calls one of the cython neuron (element *pyobject* of *status_*) functions, such as *calibrate* or *update*, the parameters contained in *status_* have to be passed to the corresponding cython neuron. The naive approach of doing that is by copying them into the cython neuron, calling the corresponding function and retry them back.
This could work for *calibare*, but not for *update* as it is called many times and this approach would slow down the system.
Furthermore, as one can realize, during a simulation the only important constantly updated parameters by the system are :

- currents
- in_spikes

- ex_spikes
- t_lag
- spike

These are called **Standard Parameters** and are the only one worth copying in the cython neuron before calling the corresponding function.
The best way of speeding up the parameter passing is by giving to the cython neuron direct pointers to these values. This is done during the calibration:

```
// Pointers to Standard Parameters passing
(*state_)[Name("pyobject")]->putStdParams(&currents,
                  &in_spikes, &ex_spikes, &t_lag, &spike);
```

Therefore, the cython neuron will change the values pointed by these pointers without making any copy, thus speeding up the system.

The update method has been highly optimized:

```
for ( long_t lag = from ; lag < to ; ++lag )
  {
*currents = B_.currents_.get_value(lag);
    *in_spikes = B_.in_spikes_.get_value(lag); // arriving in spikes
    *ex_spikes = B_.ex_spikes_.get_value(lag); // arriving ex spikes
    *t_lag = lag;

    if(this->optimized) {
        pyObj->call_update_optimized();
    }
    else {
        pyObj->call_update();
    }

    // threshold crossing
    if (*spike)
    {
      ...
    }
    ...
  }
```

There is no lookup of the *status_* dictionary as well as any unimportant test. The Standard Parameters values are directly changed.

5.3 Handling the Standard Parameters pointers

How are the Standard Parameters pointers extracted? This is done in the *putStdParams* method of the *PyObjectDatum* object:

```
void putStdParams(double** curr, double** is,
                                double** es, long** tl, long** sp) {
    ...
// numeric conversion in order to create a long variable
// containing the address of the std params
*curr = currents = PyInt_AsLong(
          PyObject_CallMethod(this->pyObj, "getPCurrents", NULL));
```

The object makes therefore two copies of the pointer, one for its own use and the other for the c++ neuron (which has to modify the values during the update). The *getPCurrents* method is contained in the *Neuron.pyx* file and looks like this:

```
cpdef getPCurrents(self):
    return <long>(&(self.currents))
```

Hence the fact that a pointer is nothing else than an integer value representing the variable address is used. This approach is mandatory since there is no possibility to directly pass a long pointer through python to c++. This is some sort of cast allowing python to see the address just like a normal integer.

5.4 Function pointers Optimization

The last optimization concerns the cython neuron function pointer itself. Instead of calling a function like *PyObject_CallMethod* in order to access the model's methods, one can extract the pointer directly:

```
PyMethodDef* updateRef = getUpdateRef(pyObj->ob_type->tp_methods);
if(updateRef != NULL) {
    this->updateFct = updateRef->ml_meth;
}
```

Now it is possible to directly call *updateFct* instead of using the Python calling mechanism. This increases the speed and has another important consequence: there isn't any need of the GIL. This latter prevents threads from accessing the same ressources. It is important when using the normal calling convension, but now the update method looks like:

```
void call_update_optimized() {
    updateFct(this->pyObj, NULL);
}
```

The drawback is that the model cannot use any python object. That means the user can only create pure C members.
The solution is therefore to provide two possibilities (see User Documentation) and calling either the previous function or this one:

```
void call_update() {
    PyGILState_STATE s = PyGILState_Ensure();
```

```
    updateFct(this->pyObj, NULL);

    PyGILState_Release(s);
}
```

5.5 Handling the parameters passing

The last task *PyObjectDatum* has to accomplish is the parameter passing during *getStatus* and *setStatus*. This is done by the *call_status_method*.

- When setting the parameters, a new python dictionary has to be created, filled with the elements of *status_* (after having been correctly translated by CyNEST) and given to the cython neuron *setStatus* method. Note that we do not want to translate every *status_* element because the cython neuron doesn't care about general parameters. The other reason is that we, of course, do not want to translate the cython neuron as well (remember it is included in the *status_* dictionary)!
- When getting the parameters, a new python dictionary is received by the *getStatus* method of the cython neuron. The content must then update the *status_* dictionary. The final step is to also include the Standard Parameters.

For the purpose of completeness, here is an example showing the *getStatus* operation:

```
// looping through the received python dictionary
while (PyDict_Next(dict, &pos, &key, &subPyObj))
{
    (**status)[PyString_AsString(key)] =
                dataConverter.objectToDatum(subPyObj);
}
// std params updating
...
```

6 Special Objects

Sometimes, a neuron model needs some special information from the system, such as simulation parameters. The values represent time units, simulation delays, ect...
Therefore, the system has to allow the final user to access these variables and this is achieved by what are called **Special Objects**.
When writing a normal C++ model, these features are provided by the *Time* and *Scheduler* classes. Therefore, a way in order to make them available from

the Python side (from the custom model) is necessary. The purpose is to emulate commands such as :

```
Scheduler::get_min_delay()

Time::get_resolution().get_ms()

Time(Time::step(steps)).get_tics()
```

The first step for emulating those classes is by defining them on the Cython side (*cynestclasses.pxd*):

```
cdef extern from "time_scheduler.h":
    cdef cppclass Time:
        long get_tics()
        long get_steps()
        ...

    cdef cppclass UnitManager:
        UnitManager(int, long)
        UnitManager(int, double)
        Time generateTime()

cdef extern from "time_scheduler.h" namespace "nest::Time":
    Time get_resolution()
    void set_resolution(double)
    ...

cdef extern from "time_scheduler.h" namespace "nest::Scheduler":
    unsigned int get_modulo(unsigned int)
    unsigned int get_slice_modulo(unsigned int)
    ...
```

The first class is the corresponding Time class, whereas the other one will be discussed soon. One can also note that static methods are defined as well. For a list of all the available methods, please consult the user documentation.

If the C++ *nestkernelnest_time.h* file is read, one can see that the Time class contains other nested classes: tic, step, ms and ms_stamp. These are different units. Now, sadly Cython cannot handle nested classes, therefore another way of extracting them is necessary. That's why the **UnitManager** class has been created. The only important function of these units is to enable the creation of a Time object, like in the following example:

```
Time::step(steps).get_tics()
```

Therefore, a UnitManager object can be created by giving it a code corresponding to a particular unit:

- tic : 1
- step : 2
- ms : 3
- ms_stamp : 4

Once the need for a Time object arises, the *generateTime* method can be called, which creates a Time object according to the internal state of the UnitManager object (which contains the previously created unit).
This is used by the wrapper cython units contained in the *cynest datamanager.pyx* file:

```
cdef class Unit:
    cdef classes.UnitManager *thisptr

    def __dealloc__(self):
        del self.thisptr

cdef class tic(Unit):
    def __cinit__(self, t):
        self.thisptr= new classes.UnitManager(1, <long>t)

    def create(self, t):
        return tic(t)

... # other units
```

The Time wrapper class uses the UnitManager in order to create a *classes.Time* object:

```
cdef class Time:
    cdef classes.Time thisptr

    def __cinit__(self, Unit t):
        self.thisptr = t.thisptr[0].generateTime()

    cdef Time getTime(self, classes.Time t):
        cdef Time tm = Time(ms(0.0))
        tm.thisptr = t
        return tm

    def create(self, t):
        return Time(t)

    def get_tics(self):
        return self.thisptr.get_tics()
```

```
    def get_steps(self):
        return self.thisptr.get_steps()

    def succ(self):
        return self.getTime(self.thisptr.succ())
    ...

# static methods

    def get_resolution(self):
        return self.getTime(classes.get_resolution())

    def set_resolution(self, d):
        classes.set_resolution(d)

    ...

cdef class Scheduler:
    def get_modulo(self, v):
        return classes.get_modulo(v)

    ...
```

Note that *t.thisptr[0]* corresponds to the UnitManager object. Both non-static and static methods are treated the same way. That makes further things easier.

Now comes a very important key idea. Since every model is automatically linked and compiled by *pyximport*, it cannot access the project definitions, therefore the custom model cannot see the Time and Scheduler classes. There is however the possibility to pass to the model a generic object and to call its methods, if they are known.
The idea is to directly pass to each model a new Time and Scheduler (as well as an object for every unit) object, so that it can access all the methods. However, since the user can normally create as many Time objects as he wants, the *create* method has been implemented. It actually creates a new object instance. We can therefore create a new instance from an existing instance and that is the key for avoiding to know the class definition.

The last step is to create wrapper classes on the model side, therefore contained in the *"installation path"includeNeuron.pyx*. In order to contain all the imported generic objects, a manager has been created:

```
cdef class ObjectManager:
    cdef object schedulerObj
```

```
    cdef object timeObj
    ...

    def setScheduler(self, obj):
        self.schedulerObj = obj

    def setTime(self, obj):
        self.timeObj = obj

    ...

cdef ObjectManager objectManager = ObjectManager()


# These methods are called from the project in order to
# fill the different imported objects

def setScheduler(obj):
    objectManager.setScheduler(obj)

def setTime(obj):
    objectManager.setTime(obj)
...
```

Note that an object of every unit has also been created (not shown here). The project side will transfer objects using, for example, the *setScheduler* function. A wrapper class for each unit is created:

```
cdef class Unit:
    cdef object ob

cdef class tic(Unit):
    def __cinit__(self, t):
        self.ob = objectManager.ticObj.create(t)
...
```

Where t represents the unit value and is given to the *create* method in order to generate a new object instance.
The Time wrapper class is very straighforward:

```
cdef class Time:
    cdef object time_ob

    def __cinit__(self, Unit t):
        self.time_ob = objectManager.timeObj.create(t.ob)

    def get_tics(self):
```

```
        return self.time_ob.get_tics()
    ...
```

Finally, in order to distinguish between static and non-static methods, a protocol has been followed, where each static method is also available as a direct function:

```
...
cpdef Time_neg_inf():
    return objectManager.timeObj.neg_inf()

cpdef Scheduler_get_modulo(v):
    return objectManager.schedulerObj.get_modulo(v)
...
```

Note that the Time_ or Scheduler_ prefixes are defined in order to specify that originally these are a static functions rather than normal instance methods.

As an example, the user can therefore translate the previously mentioned commands by writing:

```
Scheduler_get_min_delay()

Time_get_resolution().get_ms()

Time(step(steps)).get_tics()
```

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.