Tutorial

O. Rochel

September 8, 2004

Contents

1 Introduction

1.1 About this tutorial

The aim of this tutorial is to present some basic uses of the mvaspike simulator. The simulator consists of a core C++ library and a few additional utilities. In the following, the term 'simulator' will refer mainly to the library itself.

All the examples will be given in the C++ language. However, using the simulator from a different language (eg. python) or through a graphical user interface should be straightforward. In any case the same logical progression while constructing the network and running the simulation is required.

1.2 Definitions

Following the DEVS formalism, a neuron is an atomic element that sends and receives events. We also assume that these events are not valuated : the only meaningful information is the event time.

More formally, a neuron is a set

$$NI = \{s, ta, \delta_{int}, N_{in}, \delta_{ext}\}$$

where :

- $s \in S$: state variables, from a state space S.
- $x \in X$: inputs identifiers.
- $ta : S \to \mathbb{R}^+$: time advance function (time to next spike).
- $\delta_{int} : S \to S$: internal transition (reset) function.
- $\delta_{ext} : Q \times X \to S$: external transition function ($Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$).

The input identifiers will be in the following assumed to be a finite set of identifiers, or simply a finite range $[0 : N - 1]$ where N is the number of input of the neuron.

ta is used to compute the next firing time, given the present state variables and assuming no event will be received. It can be seen as a way to define the internal dynamic of the neuron. This description is completed by the δ_{int} function, which describes the state variable changes that occur at event times.

δ_{ext} describes how the state variables of a neuron change when receiving an event (a spike) through a given input port x.

A *connection* is a topological link between two elements i and i. It can be represented by a set

$$(i, y_i, j, x_j)$$

where

- $y_i \in Y_i$ is the output form we connect from
- $x_j \in X_j$ is the input port we connect to

It is often convenient to add a propagation delay d to the links, which yields a set (i, y_i, j, x_j, d).

Networks are built of neurons and connections. Additionnaly, a hierarchical or modular structure can be describes - using concepts such as subnetworks or neuronal populations. The way this is achieved is beyond the scope of the tutorial. It follows the classical way of forming hierarchical systems, such as the one given in the DEVS formalism, and ensures that the resulting systems are well-defined.

2 Modelling neurons

2.1 Choosing the right model

The most well-known spiking neuron model is undoubtedly the Leaky Integrate-and-Fire neuron (or LIF for short), which is commonly used in many modellings tasks. Its roots can be traced back to the very first models exposing the electrical properties of the nerves [Lapicque, 1907].

Unfortunatly, there exists many variations around that basic model. The simulator has a few different predefined models built-in. However, in some cases, the user will want to define other neuronal dynamics. Such a task is beyond the scope of this tutorial (XXX refer elsewhere). In the course of the tutorial, we will only use a simple type of predefined neuron : the *Lif* neuron, which is (as its name implies) a kind of leaky integrate and fire neuron.

We could also have chosen :

- The Phase model, whose dynamical properties can be linked with those of the integrate-and-fire, but from a different (more abstract) point of view.
- A more detailed integrate-and-fire, with STDP
- A simpler neuron (finite state automata)...

However, the integrate-and-fire should be sufficient to expose the basic ideas behind the simulator. Its dynamical behaviour can be written as the following :

$$\delta_{ext}(V_0, x, t) = I + (V_0 - I)\exp(-t) + w_x$$
$$\delta_{int}(V_0) = 0$$
$$ta(V_0) = \begin{cases} +\infty & \text{if } V_0 < \theta \text{ and } I \leq \theta \\ 0 & \text{if } V_0 \geq \theta \ , \\ log\left(\frac{I-V_0}{I-\theta}\right) & \text{otherwise} \end{cases}$$

To create an instance of such a neuron :

```
Lif neuron;
```

The Lif constructor does not take any argument. The neuron's parameters will be changed if necessary using various functions, as described in the next section. This behaviour is common in the library : all the models use some intuitive defaults parameters. Once declared, they can therefore be used without additional work. This helps a lot when testing or prototyping a network.

Indeed, the neuron we have just created is ready to be used. Default parameters ensure a proper functioning. They are summarized in the following :

- Initial membrane potential V=0.0
- Membrane time constant Tau=1.0
- Current I=1.58
- Firing threshold θ=1.0

The default current corresponds to a firing rate of about 1Hz.

2.2 Parameters settings

We start by describing the input of the neuron : the predefined Lif model can have N different inputs, or synapses, that will be identified by a number in the range $[0 : N-1]$.

```
Lif neuron;

neuron.set_nb_inport(2); // we want 2 different input ports
```

set_nb_inport is a method inherited from the base Devs classes. By default, a Devs model has no input port, it is the responsability of the user to declare them.

In a Lif, each synapse is associated with a weight (that models synaptic efficacy), which can be set using set_w (default value is 0.0) :

```
Lif neuron;

neuron.set_nb_inport(2);
neuron.set_w(0,0.1);     // excitatory
neuron.set_w(1,-0.2);    // inhibitory
```

We can also provide special values for the parameters that describe the core dynamical properties of an integrate and fire, such as the membrane potential and the current :

```
Lif neuron;

neuron.set_nb_inport(2);
neuron.set_w(0,0.1);
neuron.set_w(1,-0.2);

neuron.set_Tau(1.1);     // membrane potential
neuron.set_I(0.0);       // no external current
```

2.3 Testing the neuron

A typical simulation is built of at least one Discrete-Event model (here, the Lif neuron) and a root simulator that handles the task of processing events in the proper order and propagates the events between objects. A minimal program should look like the following :

```
// -- start here
#include <mvaspike.h> // our all-in-one header
int main()
{
   Lif neuron;
   Root simulator;

   simulator.set_system(neuron);
    // validation step
   simulator.init();
}
// -- end here
```

Or, including the parameters settings :

```
// -- start here
#include <mvaspike.h> // our all-in-one header
int main()
{
    Lif neuron;
    Root simulator;

    neuron.set_nb_input(1);
    neuron.set_w(0,0.25);
    neuron.set_threshold(1.0);

    simulator.set_system(neuron);

    simulator.init();
}
// -- end here
```

Now the effective simulation can start : for example, simulating one single event (one spike) can be done using :

```
simulator.step();
```

More details on how to control the simulation and collect the results will be given in section 5.

3 Modelling networks

3.1 Coupled models

The simulator can handle hierarchical models, that is, models that are grouped together to form a new model. A 'Coupled' is a modeling object in charge of this task. In order to build a network composed of two neurons, we can write :

```
// -- start here
#include <mvaspike.h>
int main()
{
    Lif n1,n2;
    Root simulator;
    Coupled cpl;

    cpl.add_component(n1);
    cpl.add_component(n2);
    simulator.set_system(cpl);

    simulator.init();
}
// -- end here
```

Notice that now the root simulator handle the simulation of the 'Coupled' object. Note that the two neurons here are not connected. Therefore, they could have been simulated independently : the only task for the Root simulator is here to order (interleave) the firing times of both neurons.

3.2 Connecting objects

```
// -- start here
#include <mvaspike.h>
int main()
{
    Lif n1,n2;
    int i1,i2;
    Root simulator;
    Coupled cpl;

    n2.set_nb_inport(1);
    n2.set_w(0,0.2);
    i1=cpl.add_component(n1);
    i2=cpl.add_component(n2);
    cpl.connect(i1,0,i2,0);
    simulator.set_system(cpl);

    simulator.init();
}
// -- end here
```

The connect function creates a topological link between an output port of a component to an input port of another component. Since neurons have only one output, the source port is 0 here. We have declared here that the neuron n2 has only one input port (associated with the weight 0.2), so the destination port is also 0.

In order to identify the component, you can safely assume that their identifiers correspond to the order in which they have been added to the Coupled element. These numbers are also conveniently returned by add_component, as shown in the example.

3.3 Global vs. local identifiers : numbering the world

In the previous section, the connect function makes use of some numbers to identify the neurons one want to connect from or to. This identifier is local, since it only permits to identify a sub-component within a coupled component. As a consequence, it should be emphazised that it is not possible to create a connection directly between an object in one coupled component and an external object. A global identifier is however associated with every element in the system. It can be used for example to easily record events from different part of the hierarchy (see section §4.2).

3.4 An easier way to build simple networks

In many cases, a network will present some kind of homogeneity. It is for example very common to build networks composed of one unique model of neuron. Some objects have therefore been implemented that permit the definition of sets of identical neurons. The SimpleNet object is one of these. Moreover, as a typical simple network is composed of a single population, the SimpleNet object embeds also the root simulator, so as to permit the declaration of a network with only one line of code.

The user is however obviously in charge of properly setting all the parameters, initial state values and connections within the network. Additionally, since no Recorder is associated with it, the user will have to declare how to collect the results.

```
// -- start here
#include <iostream>
using namespace std;

#include "../mvaspike.h"

#define N 20
int main()
{
    SimpleNet<Lif> net(N);

    for (int i=0;i<2;i++)
        {
            net[i].set_nb_inport(1);
            net[i].set_w(0,0.2);
            net.connect(i,(i+1)%N,0,0.0);
        }
  net[0].set_V(1.0);

  net.init();

  net.iter();
  // we can check the current simulation time now
  // (should be t=0.0)
  cout << "Time ="<< net.get_t() << endl;
  // In a simplenet, connections are always delayed
  // (but the delay can be 0.0 as here), so
  // within an iteration, only receptions XOR firings
  // can appear. We thus need a second call to iter()
  // to ensure the receptions can occur
  net.iter();
  // (should be t=0.0)
  cout << "Time ="<< net.get_t() << endl;
  // (V2 should be = 0.2)
  cout << "V2   ="<< net[1].get_V() << endl;

  // a second iteration now :
  net.iter();
  // the second neuron will fire earlier that its spontaneous period
  // due to the excitatory contribution at t=0
  // (time should be < 1.0)
  cout << "Time ="<< net.get_t() << endl;
  return 0;
}
```

4 Setting up the simulation

4.1 Initialization

The initialization step is needed internally to ensure the correctness of the declared system (eg. a connection to an input port requires that this particular input port has been declared somewhere). This is also an optimization phase : the data is reorganized internally to ensure a fast simulation and efficient memory use. As it is mostly an internal step, one could have chosen to hide it from

the user. However, it has been kept visible as a remainder of the differences between modeling and simulation steps.

4.2 Recorders

Every event that occurs during the simulation can be recorded using Recorder objects. The existing Recorders differ in the way they store the data (in memory, on a IOstream, ...), as well as in the kind of events they record or just drop. For example, a SpikeRecorder will only record spikes : the reception times, even the delayed ones, will not be recorded.

As the simulator is oriented towards the simulation of hierarchical systems, the output of the simulations must take care of the origin of the events within the hierarchy. Different solutions can be used :

- store a global identifier with each event record. This identifier can then be used to retrieve the node of the hierarchy that created the event. This is the simpler behaviour, and the default one in many recorders.
- store a path from the root of the hierarchy to the active element within each event record. It is a variation of the previous solution, as the path can be interpreted as a global identifier. It permits a more immediate readout of the results (eg. during an interactive simulation session), but is however more verbose and therefore less memory-efficient. The identifiers appearing in the paths are the local identifiers for each node, from the root element to the active node.
- assign a different recorder to each node of interest. One can record that way a small subset of the overall activity, and direct the output to different files for example.

The most simple recorde gives a verbose output, probably useful for debugging only. Here, we will describe the SpikeRecorder, which is probably the more useful recorder.

```
#include <iostream>
using namespace std;

// the main API header file
#include <mvaspike.h>

#define N 2
int main()
{
  SimpleNet<Lif> net(N);
  SpikeRecorder rec;                  // create a recorder
  for (int i=0;i<2;i++)
    {
      net[i].set_nb_inport(1);
      net[i].set_w(0,0.07);
      net.connect(i,(i+1)%N,0,0.0);
    }

  net[0].set_V(0.2);
  net[0].set_V(0.5);
  net.record(rec);                    // <- register the recorder
  net.init();

  net.run(10.0);                      // run for 10s
  return 0;
}
```

The output should ressemble :

```
1 0.620115
2 0.914154
1 1.55890
2 1.82607
1 2.49937
2 2.73532
1 3.44171
2 3.64136
1 4.38617
2 4.54354
1 5.33299
2 5.44102
1 6.28244
2 6.33271
1 7.23479
2 7.23479
1 8.18954
2 8.18954
1 9.14428
2 9.14428
```

Each line is composed of an global identifier (1 or 2 here) and an event time, meaning that a spike was emitted by the corresponding neuron at that time.

TODO : local / set_id_mode etc.

5 Controlling the simulation

5.1 Step by step simulation

Note that a 'step' in our discrete-event based approach really means an event : the simulator will find the next event pending, and process it.

5.2 Batch runs

6 Technical stuff

6.1 Installing the package

The library is packaged using the very common GNU installation toolchain (autoconf/automake/libtool).

Once decompressed/detarred/unzipped or whatever is your initial source, go into the root directory, and issue the following commands :

```
./configure
make
make install
```

It will compile the library and install it in /usr/local/lib/ by default (the include files will go into /usr/local/include/mvaspike). If you prefer a different root installation directory, you can pass it to the configure script like this :

```
./configure --prefix=/my/preferred/installation/path
```

The library will then be installed in /my/preferred/installation/path/lib and the include files accordingly in /my/preferred/installation/path/include/mvaspike.

The configure script has a bunch of other different options : one can list them using

```
./configure --help
```

(or ask your local GNU guru)

6.2 Compilation

The library does not require any exotic external dependency. Compiling should be straightforward, eg under linux :

```
% gcc tutorial1.cc -o tutorial -lmvaspike
```

should be sufficient, provided all .h and libraries are reachable using the include and library paths.

7 What to do next...

Right now you probably want to have a coffee break, but when you come back, here are some possible continuations :

- Have a look at the different examples, which should be almost readable now. They are commented.
- Read the reference manual, which exposes all the differents objects we can use and their parameters.

References

[Lapicque, 1907] Lapicque, L. (1907). Recherches quantitatives sur l'excitation lectrique des nerfs traits comme une polarisation. *J. Physiol. Pathol. Gen, 9, pp. 620–635.*