pyramidal Documentation

Release 0.0.1

Mike Vella

September 29, 2014

Contents

CHAPTER 1

A simulator-independent API for multi-compartmental modelling of neurons

Contents:

1.1 What it does

Pyramidal is a simulator-independent API for modelling of multicompartmental neurons. The basic problem addressed is interoperability and exchange of multicompartmental neuron models.

Pyramidal is designed to utilise an object model provided by an efficient libNeuroML backend.

1.2 Installing pyramidal

pyramidal has the following strong dependencies:

- NEURON (Python module, for installation instructions see http://www.davison.webfactional.com/notes/installation-neuron-python/)
- pyMOOSE (for installation instructions see below)
- libNeuroML (for installation see below)
- numpy (http://numpy.scipy.org/)

1.2.1 pyMOOSE instllation

The latest version of MOOSE with a Python interface can be installed as follows:

```
svn co http://moose.svn.sourceforge.net/svnroot/moose/moose/branches/dh_branch moose
cd moose
make pymoose
sudo cp -r python/moose /usr/lib/python2.7/dist-packages
```

1.2.2 Get a read only copy of libNeuroML

Install git and type:

```
git clone git://github.com/NeuralEnsemble/libNeuroML.git
```

More details about the git repository and making your own branch/fork are here.

1.2.3 Install libNeuroML

Use the standard install method for Python packages:

```
sudo python setup.py install
```

1.2.4 Get a read only copy of pyramidal

```
git clone https://github.com/vellamike/pyramidal.git
```

1.2.5 Install pyramidal

Use the standard install method for Python packages:

```
sudo python setup.py install
```

1.3 Note on units

Unless otherwise stated, all units in pyramidal documentation and examples are as follows:

- cm - uF/cm^2
- ra - mmho/cm2
- rm - cm^2/mmho
- g_channel - mmho/cm^2 (e.g g_na)
- length - um
- voltage - mV

1.4 Examples

All these examples are located in the /examples folder

1.4.1 Example 1 - A passive neuron

This example serves as an introduction to pyramidal usage. Let's jump straight in and examine the code.

We have the following interesting import statements:

```
import neuroml.morphology as ml
import neuroml.kinetics as kinetics
import pyramidal.environments as envs
```

- The neuroml morphology module provides support for dealing with compartments - their dimensions and their connectivity.
- The neuroml kinetics module provides objects with a time-dependent element to their behaviour. This includes things like point currents.
- The pyramidal environments module provides support for interfacing with different simulators. An "environment" in this context can be thought of an entire simulator contained within an object.

The syntax for making a compartment is intuitive:

```
compartment = ml.Segment(length=500,
                         proximal_diameter=500,
                         distal_diameter=500)
```

The PassiveProperties and LeakCurrent objects are created, which we will then associate with our compartment.

```
#Create a PassiveProperties object:
passive = kinetics.PassiveProperties(init_vm=-0.0,
                                     rm=1/0.3,
                                     cm=1.0,
                                     ra=0.03)

#Create a LeakCurrent object:
leak = kinetics.LeakCurrent(em=10.0)
```

The morphology of a compartment is an object with all the other compartments connected to it. So for instance, if I had a compartment 'iseg' which was part of a much larger morphology including an axon and a whole cell, iseg.morphology would be the whole morphology. We need this object to pass to our simulators later on. Additionally, as shown here, the passive properties and leak current are passed to a whole morphology. It is currently not possible to set these per compartment (though this feature can be introduced if there is demand for it).

```
#Get the Morphology object which the compartment is part of:
morph = compartment.morphology

#insert the passive properties and leak current into the morphology:
morph.passive_properties = passive
morph.leak_current = leak
```

We now use the neuroml kinetics module to create a current clamp stimulus. Otherwise the simulation would be a very boring one indeed. We insert the stimulus into the morphology, note that unlike with the passive and leak currents which were inserted into the morphology as a whole, the current clamp, being a point current, is inserted into a segment,hence the morph[0] statement - this means the first segment in the morphology (in our case slightly irrelevant as the morphology only contains one segment anyway)

```
#create a current clamp stimulus:
stim = kinetics.IClamp(current=0.1,
                       delay=5.0,
                       duration=40.0)

morph[0].insert(stim)
```

All that now remains is to create the MOOSE and NEURON environments and run the simulations, the syntax is the same for both environments. Here's MOOSE:

```
#Create the MOOSE environmet:
moose_env = envs.MooseEnv(sim_time=100,
                          dt=1e-2)

#import morphology into environment:
```

```
moose_env.import_cell(morph)

#Run the MOOSE simulation:
moose_env.run_simulation()
```

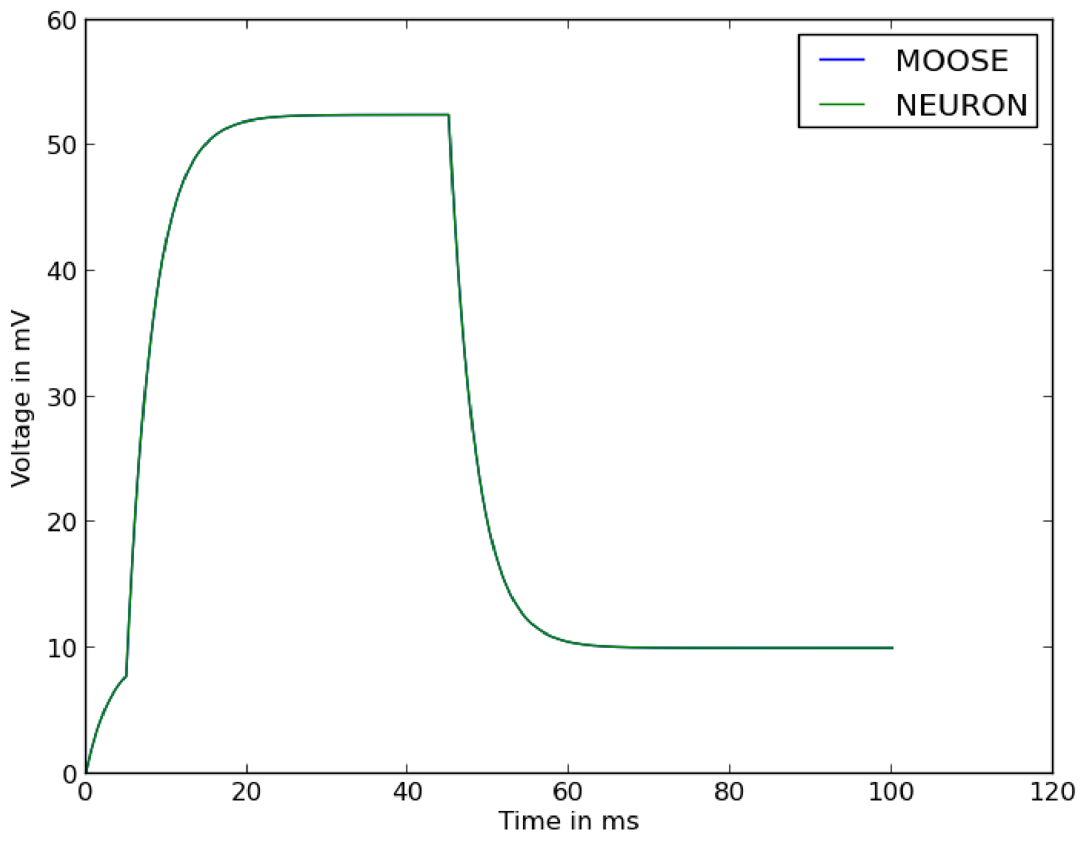
And NEURON:

```
#create the NEURON environment
neuron_env = envs.NeuronEnv(sim_time=100,
                            dt=1e-2)

#import morphology into environment:
neuron_env.import_cell(morph)

#run the NEURON simulation
neuron_env.run_simulation()
```

The final part of the example simply runs some plotting routines to visualise the result of our simulation. Assuming the example runs correctly, you should get a plot looking like this:



As can be seen, the result of this passive, single-compartment similation are so similar in NEURON and MOOSE that it is almost impossible to tell there is more than one plot.

1.4.2 Example 2 - Hodgkin-Huxley single compartmental simulations

This example is the same as Example 1 except that we are now going to use the neuroml kinetics module to add some Hodgkin-Huxely channels. This will allow us to see some action potentials! The libNeuroML type "HHChannel" is used to insert some kinetics. Appropriately, this example roughly recreates Hodgkin and Huxley Squid giant axon model.

Just as in example 1 we created a current clamp stimulus, we are now going to create sodium and potassium ion channel objects:

```
#create Na ion channel:
na_channel = kinetics.HHChannel(name = 'na',
                                specific_gbar = 120.0,
                                ion = 'na',
                                e_rev = 115.0, #115 for squid
                                x_power = 3.0,
                                y_power = 1.0)

#create K ion channel:
k_channel = kinetics.HHChannel(name = 'kv',
                               specific_gbar = 36.0,
                               ion = 'k',
                               e_rev = -12.0,
                               x_power = 4.0,
                               y_power = 0.0)
```

Here the xpower and ypower signify the power to which activating("m") and inactivating("h") components of the channel should be raised. The next thing we need to do is set the coefficients which determine the gating parameters governing the alpha and beta opening and closing rates of each gate, the snippet here is for the sodium m gate:

```
na_m_params = {'A_A':0.1 * (25.0),
               'A_B': -0.1,
               'A_C': -1.0,
               'A_D': -25.0,
               'A_F':-10.0,
               'B_A': 4.0,
               'B_B': 0.0,
               'B_C': 0.0,
               'B_D': 0.0,
               'B_F': 18.0}
```

In terms of the Hodgkin Huxley formalism, these parameters have the following meaning:

Once these parameters have beend decided, the setup_alpha method is run on each channel, specifying whether coefficients for the X or Y gate are being set, in this snippet we do this for the activating and inacticvating gates:

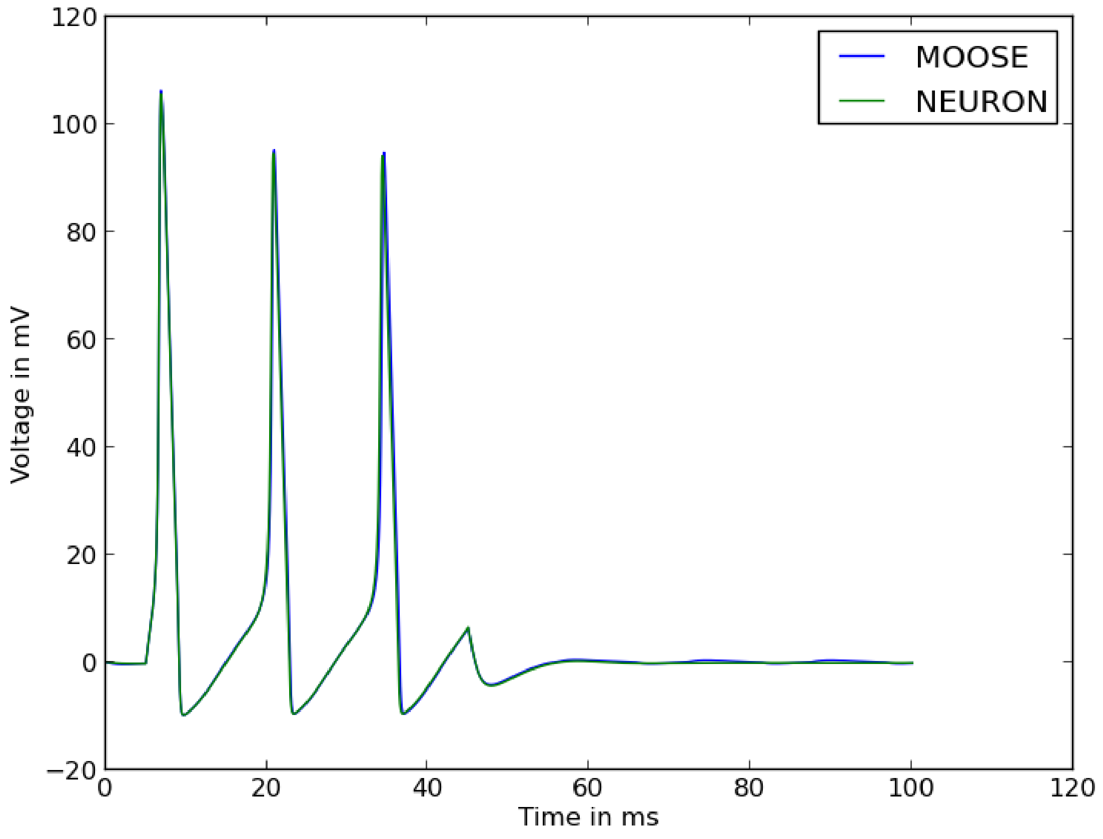
```
#setup the channel gating parameters:
na_channel.setup_alpha(gate = 'X',
                       params = na_m_params,
                       vdivs = 150,
                       vmin = -30,
                       vmax = 120)

na_channel.setup_alpha(gate = 'Y',
                       params = na_h_params,
                       vdivs = 150,
                       vmin = -30,
                       vmax = 120)
```

Just like the current clamp stimulus in example 1, the ion channel is inserted into a specific segment:

```
morphology[0].insert(na_channel)
morphology[0].insert(k_channel)
```

The MOOSE and NEURON environments are created and morphology imported just as before. Once the simulation is run you should get a plot looking something like this:



1.4.3 Example 3 - Multi compartmental morphology construction

Note: As of 21/08/12 mutli-compartmental modelling is still buggy. This is because of some unresolved issues in setting axial resistance which will soon be resolved.

This example demonstrates:

- Loading morphologies from neuroml files
- Constructing morphologies “by hand” from compartments
- Combining loaded morphologies with “hand made” ones, typically for adding an axon to a morphology or “fixing” incomplete morphologies.

First an axon is created by creating each segment and attaching them in the right order:

```
#Create an axon:
iseg=ml.Segment(length=10,proximal_diameter=1,distal_diameter=2)
myelin1=ml.Segment(length=100,proximal_diameter=3,distal_diameter=4)
node1=ml.Segment(length=10,proximal_diameter=5,distal_diameter=6)
myelin2=ml.Segment(length=100,proximal_diameter=7,distal_diameter=8)
node2=ml.Segment(length=10.0,proximal_diameter=9,distal_diameter=10)

#attatch all the segments together, in the right order:
iseg.attach(myelin1)
myelin1.attach(node1)
node1.attach(myelin2)
myelin2.attach(node2)
```

A cell is loaded from a neuroml file, the cells attribute in the doc object is a list of cell objects.

```
#load a cell from a neuroml file
doc = loaders.NeuroMLLoader.load_neuroml('./Purk2M9s.nml')
cell = doc.cells[0]
morph = cell.morphology
```

The soma (assumed to be located at index == 0 in the morphology) is attached to the initial segment:

```
#attach iseg to the soma of loaded cell:
morph[0].attach(iseg)
```

In order to get the new morphology the following needs to be done:

```
#obtain the new morphology object:
new_morphology=morph.morphology
```

This is because the morph object still refers to the loaded morphology, these objects are not automatically updated with new segments when an attach method is run but always refer to the segments which they contained at initialization.

1.4.4 Example 4 - Working with mod files

It is still possible to use mod files as long as you are working only with the NEURON environment. As with the Python library for NEURON, the mod files need to be in the top-level directory where your simulation script is located. The first thing you need to do is run the nrnivmodl command:

```
$nrnivmodl
```

Assuming the above has been done, a channel object is made, and all settable attributes (such as gbar) are modified from their default value as follows:

```
kv_attributes = {'gbar':10000}
kv = kinetics.Nmodl('kv',kv_attributes)
```

channels are then inserted in the usual way:

```
morphology[0].insert(kv)
morphology[0].insert(na)
```

It should be noted that pyramidal uses this, along with code-generation of mod files, to create channels from componetns such as HHChannel objects. A planned future update is to automate the mod file compilation stage.

1.5 Project Status (as of 21/8/12)

1.5.1 libNeuroML

libNeuroML (http://libneuroml.readthedocs.org/en/latest/index.html) is a major component of the pyramidal effort, providing the object-model backend. Currently libNeuroML has the following status.

- Read neuroml morphology - 100%
- Write neuroml - 50%
- Read HDF5 - 0%
- Write HDF5 - 100%
- Internal morphology representation - 100%
- Internal kinetics (voltage clamp, ion currents etc) representation - 50%

1.5.2 NEURON environment

- Inserting kinetic components (IClamp, VClamp etc..) into segments - 50%
- Inserting ion channels into segments - 100%
- NEURON internal representation of morphology - 100%

1.5.3 MOOSE environment

- Inserting kinetic components (IClamp, VClamp etc..) into segments - 50%
- Inserting ion channels into segments - 100%
- NEURON internal representation of morphology - 75%

CHAPTER 2

Source documentation

2.1 pyramidal

2.1.1 pyramidal Package

`environments` Module

`moosecomponents` Module

`neuronutils` Module

class `neuronutils.`**`HHNMODLWriter`**(*hh_channel*)

Bases: `neuronutils.NMODLWriter`

`write`()

class `neuronutils.`**`NMODLWriter`**

Bases: `object`

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*