Patch

Release 1.2.0

Robin De Schepper

Mar 23, 2021

CONTENTS:

CHAPTER

ONE

USAGE

Be aware that the interface is currently incomplete, this means that most parts are still "just" NEURON. I've only patched holes I frequently encounter myself when using the `h.Section`, `h.NetStim` and `h.NetCon` functions. Feel free to open an issue or fork this project and open a pull request for missing or broken parts of the interface.

1.1 Philosophy

Python interfaces should be Pythonic, this wrapper offers just that:

- Full Python objects: each wonky C-like NEURON object is wrapped in a full fledged Python object, easily handled and extended through inheritance.
- Duck typed interface: take a look at the magic methods I use and any object you create with those methods present will work just fine with Patch.
- Correct garbage collection, objects connected to eachother don't dissapear: Objects that rely on eachother store a reference to eachother. As is the basis for any sane object oriented interface.

1.2 Basic usage

Use it like you would use NEURON. The wrapper doesn't make any changes to the interface, it just patches up some of the more frequent and ridiculous gotchas.

Patch supplies a new HOC interpreter `p`, the `PythonHocInterpreter` which wraps the standard HOC interpreter `h` provided by NEURON. Any objects returned will either be `PythonHocObject`'s wrapping their corresponding NEURON object, or whatever NEURON returns.

When using just Patch the difference between NEURON and Patch objects is handled transparently, but if you wish to mix interpreters you can transform all Patch objects back to NEURON objects with `obj.__neuron__()` or the helper function `patch.transform`.

```
from patch import p, transform
import glia as g

section = p.Section()
point_process = g.insert(section, "AMPA")
stim = p.NetStim()
stim.start = 10
stim.number = 5
stim.interval = 10
```

(continues on next page)

(continued from previous page)

```
# And here comes the magic! This explicitly defined connection
# isn't immediatly garbage collected! What a crazy world we live in.
# Has science gone too far?
p.NetCon(stim, point_process)

# It's fully compatible using __neuron__
from neuron import h
nrn_section = h.Section()
nrn_section.connect(transform(section))
nrn_section.connect(section.__neuron__())
```

CHAPTER

TWO

SECTIONS

Sections are cilindrical representations of pieces of a cell. They have a length and a diameter. Sections are the main building block of a simulation in NEURON.

You can use the `.connect` method to connect `Sections` together.

Sections can be subdivided into `Segments` by specifying `nseg`, the simulator calculates the voltage for each segment, thereby affecting the spatial resolution of the simulation. The position of a segment is represented by its normalized position along the axis of the Segment. This means that a Segment at x=0.5 is in the middle of the Section. By default every section consists of 1 segment and the potential will be calculated for 3 points: At the start (0) and end (1) of the section, and in the middle of every segment (0.5). For 2 segments the simulator would calculate at 0, 0.333..., 0.666... and 1.

Patch

```
from patch import p
s = p.Section()
s.L = 40
s.diam = 0.4
s.nseg = 11

s2 = p.Section()
s.connect(s2)
```

NEURON

```
from neuron import h
s = h.Section()
s.L = 40
s.diam = 0.4
s.nseg = 11

s2 = h.Section()
s.connect(s2)
```

2.1 Retrieving segments

Sections can be called with an x to retrieve the segment at that x. The segments of a Section can also be iterated over.

Patch

```
s.nseg = 5
seg05 = s(0.5)
print(seg05)
for seg in s:
    print(seg)
```

NEURON

```
s.nseg = 5
seg05 = s(0.5)
print(seg05)
for seg in s:
    print(seg)
```

2.2 Recording

You can tell Patch to record the membrane potential of your Section at one or multiple locations by calling the `.record` function and giving it an `x`. If `x` is omitted `0.5` is used.

In NEURON you'd have to create a `Vector` and keep track of it somewhere and find a way to link it back to the Section it recorded, in Patch a section automatically stores its recording vectors in `section.recordings`.

Patch

```
s.record(x=1.0)
```

NEURON

```
v = h.Vector()
v.record(s(1.0))
all_recorders.append(v)
```

2.3 Position in space

With Patch it's very straightforward to define the 3D path of your Section through space. Call the `.add_3d` function with a 2D array containing the xyz data of your points. Optionally, you can pass another array of diameters.

Patch

```
s.add_3d([[0, 0, 0], [2, 2, 2]], diameters)
```

NEURON

```
s.push()
points = [[0, 0, 0], [2, 2, 2]]
for point, diameter in zip(points, diameters):
    h.pt3dadd(*point, diameter)
h.pop_section()
```

CHAPTER

THREE

CONNECTING COMPONENTS

3.1 Connecting sections

3.1.1 To other sections

Connecting sections together is the basic way of constructing cells in NEURON. You can do so using the `Section.connect` method.

Patch

```
from patch import p
s = p.Section()
s2 = p.Section()
s.connect(s2)
```

NEURON

```
from neuron import h
s = h.Section()
s2 = h.Section()
s.connect(s2)
```

3.2 Network connections

TO DO

3.2.1 In parallel simulations

In Patch most of the parallel context is managed for you, and you can use the `interpreter.PythonHocInterpreter.ParallelCon()` method to either connect an output (cell soma, axons, ...) to a GID or a GID to an input (synapse on postsynaptic cell, ...)

Patch

Detecting the spikes of a Section and connecting them to GID 1:

```
from patch import p
gid = 1
s = p.Section()
nc = p.ParallelCon(s, gid)
```

Connecting the spikes of GID 1 to a synapse:

```
from patch import p
gid = 1
syn = p.Section().synapse(p.SynExp)
nc = p.ParallelCon(gid, syn)
```

NEURON

Detecting the spikes of a Section and connecting them to GID 1:

```
from neuron import h
gid = 1
h.nrnmpi_init()
pc = h.ParallelContext()
s = h.Section()
nc = h.NetCon(s(0.5)._ref_v, None)
pc.set_gid2node(gid, pc.id())
pc.cell(gid, nc)
pc.outputcell(gid)
```

Connecting the spikes of GID 1 to a synapse:

```
from neuron import h
gid = 1
h.nrnmpi_init()
pc = h.ParallelContext()
s = h.Section()
syn = h.SynExp(s)
pc.gid_connect(gid, syn)
```

CHAPTER

FOUR

MAGIC METHODS

4.1 __neuron__

Get the object's NEURON pointer

Whenever an object with this method present is sent to the NEURON HOC interpreter, the result of this method is passed instead. This allows Python methods to encapsulate NEURON pointers transparently

4.2 __netcon__

Get the object's NetCon pointer

Whenever an object with this method present is used in a `NetCon` call, the result of this method is passed instead. The connection is stored on the original object. This allows to simplify the calls to NetCon, or to add more elegant default behavior. For example inserting a connection on a section might connect it to a random segment and you'd be able to use `p.NetCon(section, synapse)`.

CHAPTER

FIVE

PATCH PACKAGE

CHAPTER

SIX

CANDY GUIDE

6.1 Referencing

All objects in Patch that depend on each other reference each other, meaning that Python won't garbage collect parts of your simulation that you explicitly defined. An example of this is that when you create a synapse and a NetStim and connect them together with a NetCon, you only have to hold on to 1 of the variables for Patch to know that if you're holding on to a NetCon you are *most likely* interested in the parts it is connected to. There is no longer a need to store every single NEURON variable in a list or on an object somewhere. This drastically cleans up your code.

6.2 Sections & Segments

- When connecting Sections together they all reference each other.
- Whenever you're using a Section where a Segment is expected, a default Segment will be used (Segment 0.5):

```
s = p.Section()
syn = p.ExpSyn(s)
# syn = h.ExpSyn(s(0.5))
```

- Whenever a Section/Segment is recorded or connected and a NEURON scalar is expected `_ref_v` is used:

```
s = p.Section()
v = p.Vector()
v.record(s) # v.record(s(0.7)._ref_v)
```

- Sections can record themselves at multiple points:

```
s = p.Section()
s.record() # Creates a vector, makes it record s(0.5)._ref_v and stores it as a␣
↪recorder
s.record(0.7) # Records s(0.5)._ref_v

plt.plot(list(s.recordings[0.5]), list(p.time))
```

- Sections can connect themselves to a PointProcess target with `.connect_points`, which handles NetCon stack access transparently. This allows for example for easy creation of synaptic contacts between a Section and a target synapse:

```
s = p.Section()
target_syn = p.Section().synapse(p.ExpSyn) # Creates an ExpSyn synapse
s.connect_points(target_syn) # Creates a NetCon between s(0.5)._ref_v and target_syn
```

- Create a current clamp in a Section and configure it with keyword arguments:

```
clamp = p.Section().iclamp(amp=10, delay=0, duration=100)
# Pass an array to inject a varying current per timestep starting from the delay.
clamp2 = p.Section().iclamp(amp=[i for i in range(1000)], delay=100)
```

- You can place Sections on the stack with `.push()`, `.pop()` or a context manager:

```
s = p.Section()
s.push()
s.pop()
with s.push():
  s_clamp = p.IClamp()
# `s` is automatically popped from the stack when the context is exited.
```

- *p.SectionRef* can be called with either an arg or `sec` kwarg:

```
s = p.Section()
sr = p.SectionRef(sec=s)
sr = p.SectionRef(s)
```

6.3 Parallel networks

When you get to the level of the network the work becomes alot easier if you can describe your cells in a more structured way, so be sure to check out Arborize.

If you want to stay vanilla Patch still has you covered; it comes with out-of-the-box parallelization. Introducing the transmitter-receiver pattern:

```
if p.parallel.id() == 0:
  transmitter = ParallelCon(obj1, gid)
if p.parallel.id() == 1:
  receiver = ParallelCon(gid, obj2)
```

Just these 2 commands will create a transmitter on node 0 that broadcasts the spikes of `obj1` (sections, segments) with the specified GID and a receiver on node 1 for `obj2` (synapses, most likely?) that listens to spikes with that GID. Know that under the hood it needs to be something that can be connected to a `NetCon`.

That's it. You are now spiking in parallel!

6.3.1 Arbitrary data broadcasting

MPI has a great feature, it allows broadcasting data to other nodes. In NEURON this is restricted to just Vectors. Patch gives you back the freedom to transmit arbitrary data. Anything that can be pickled can be transmitted:

```
data = None # It's important to declare your var on all nodes to avoid NameErrors
if p.parallel.id() == 12:
  data = np.random.randint((12,12,12))
received = p.parallel.broadcast(data, root=12)
```

CHAPTER

SEVEN

INSTALLATION

Patch can be installed using:

```
pip install nrn-patch
```

CHAPTER

EIGHT

SYNTACTIC SUGAR & QUALITY OF LIFE

This wrapper aims to make NEURON more robust and user-friendly, by patching common gotcha's and introducing sugar and quality of life improvements. For a detailed overview of niceties that will keep you sane instead of hunting down obscure bugs, check out the *Candy Guide*.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search