Patch

Release 1.2.0

Robin De Schepper

Mar 28, 2022

CONTENTS:

1 Usage **3**
- 1.1 Philosophy . . . 3
- 1.2 Basic usage . . . 3

2 Sections **5**
- 2.1 Retrieving segments . . . 5
- 2.2 Recording . . . 6
- 2.3 Position in space . . . 6
- 2.4 Full reference . . . 6

3 Connecting components **9**
- 3.1 Connecting sections . . . 9
- 3.2 Network connections . . . 9

4 Magic methods **11**
- 4.1 __neuron__ . . . 11
- 4.2 __netcon__ . . . 11

5 patch package **13**
- 5.1 patch.core module . . . 13
- 5.2 patch.interpreter module . . . 14
- 5.3 patch.objects module . . . 15
- 5.4 Module contents . . . 18

6 Candy Guide **19**
- 6.1 Referencing . . . 19
- 6.2 Sections & Segments . . . 19
- 6.3 Parallel networks . . . 20

7 Installation **23**

8 Syntactic sugar & quality of life **25**

9 Indices and tables **27**

Python Module Index **29**

Index **31**

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 NEURON's `Section`, `NetStim` and `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 many 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 `neuron.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 *`__neuron__()`* or the helper function *`transform()`*.

```python
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
```

(continues on next page)

(continued from previous page)

```
stim.interval = 10

# 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.

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

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

```
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.

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

```
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`.

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

```
      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.

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

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

2.4 Full reference

Here is a full list of methods that Patch patched or added to the interface of `NEURON Sections`:

class `patch.objects.`**`Section`**(**args*, ***kwargs*)

`add_3d`(*points*, *diameters=None*)

Add new 3D points to this section xyz data.

Parameters

- **`points`** – A 2D array of xyz points.
- **`diameters`** (*`float or array`*) – A scalar or array of diameters corresponding to the points. Default value is the section diameter.

`connect`(*target*, **args*, ***kwargs*)

Connect this section to another one as child section.

`connect_points`(*target*, *x=None*, ***kwargs*)

Connect a Segment of this Section to a target. Usually used to connect the membrane potential to a point process.

`iclamp`(*x=0.5*, *delay=0*, *duration=100*, *amplitude=0*)

Create a current clamp on the section.

Parameters

- **`x`** (*`float`*) – Location along the segment from 0 to 1.
- **`delay`** (*`float`*) – Duration of the pre-step holding interval, from *0* to *delay* ms.
- **`duration`** (*`float`*) – Duration of the step interval, from *delay* to *delay* + *duration* ms.
- **`amplitude`** (*`Union[float, List[float]]`*) – Can be a single value to define the current during the step (*delay* to *delay* + *duration* ms), or a sequence to play after *delay* ms. This will play 1 value of the sequence into the clamp per timestep.

Returns The current clamp placed in the section.

Return type *`objects.SEClamp`*

`insert`(**args*, ***kwargs*)

Insert a mechanism into the Section.

`property parent`

Returns the parent of the Section, or `None`

`property points`

Return the 3d point information associated to this section.

`pop`()

Pop this section off the section stack.

`push`()

Return a context manager that pushes this Section onto the section stack and takes it off when the context is exited.

`record`(*x=None*)

Record the Section at a certain point.

Parameters **`x`** (*`float`*) – Arcpoint, defaults to `__arc__` if omitted.

`set_dimensions`(*length*, *diameter*)

Set the length and diameter of the piece of cable this Section will represent in the simulation.

`set_segments`(*segments*)

Set the number of discrete points where equations are solved during simulation.

`synapse`(*factory*, **args*, *store=False*, ***kwargs*)

Insert a synapse into the Section.

Parameters

- **`factory`** (*`callable`*) – Callable that creates a point process, is given the Section as first argument and passes on all other args.
- **`store`** (*`bool`*) – Store the synapse on the Section in a `synapses` attribute.

`vclamp`(*x=0.5*, *delay=0*, *duration=100*, *after=0*, *voltage=- 70*, *holding=- 70*)

Create a voltage clamp on the section.

Parameters

- **`x`** (*`float`*) – Location along the segment from 0 to 1.
- **`delay`** (*`float`*) – Duration of the pre-step holding interval, from *0* to *delay* ms.
- **`duration`** (*`float`*) – Duration of the step interval, from *delay* to *delay* + *duration* ms.
- **`after`** (*`float`*) – Duration of the post-step holding interval, from *delay* + *duration* to *delay* + *duration* + *after* ms.
- **`voltage`** (*`Union[float, List[float]]`*) – Can be a single value to define the voltage during the step (*delay* to *delay* + *duration* ms), or 3 values to define the pre-step, step and post-step voltages altogether.
- **`holding`** (*`float`*) – If *voltage* is a single value, *holding* is used for the pre-step and post-step voltages.

Returns The single electrode voltage clamp placed in the section.

Return type *`objects.SEClamp`*

`wholetree`()

Return the whole tree of child Sections

Return type List[patch.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.

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

```
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 `ParallelCon()` method to either connect an output (cell soma, axons, . . .) to a GID or a GID to an input (synapse on postsynaptic cell, . . .).

The following code transmits the spikes of a Section on GID 1:

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

```
from neuron import h
gid = 1
h.nrnmpi_init()
```

(continues on next page)

(continued from previous page)

```
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)
```

You can then receive the spikes of GID 1 on a synapse:

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

```
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

5.1 patch.core module

`patch.core.`**`assert_connectable`**(*obj*, *label=None*)

Assert whether an object could be used as a `Connectable`.

Parameters **`label`** (`str`) – Optional label to display to describe the object if the assertion fails.

`patch.core.`**`is_point_process`**(*name*)

Check if a PointProcess with `name` exists on the `HocInterpreter`.

Parameters **`name`** (`str`) – Name of the PointProcess to look for. Needs to be a known attribute of `neuron.h`.

Returns Whether an attribute with `name` exists on `neuron.h` and has functions matching those expected to be present on a `PointProcess`.

Return type bool

`patch.core.`**`is_section`**(*obj*)

Check if the class name of an object is `Section`.

`patch.core.`**`transform`**(*obj*)

Transform an object to its NEURON representation, if the `__neuron__` magic method is present.

`patch.core.`**`transform_arc`**(*obj*, **args*, ***kwargs*)

Get an arclength object on a NEURON object. Calls the `__arc__` magic method on the callable object if present, otherwise returns the transformed object.

`patch.core.`**`transform_netcon`**(*obj*)

Transform an object into the pointer that should be connected, if the `__netcon__` magic method is present.

`patch.core.`**`transform_record`**(*obj*)

Transform an object into the pointer that should be recorded, if the `__record__` magic method is present.

5.2 patch.interpreter module

class patch.interpreter.**ParallelContext**(**args*, ***kwargs*)

Bases: *patch.objects.PythonHocObject*

broadcast(*data*, *root=0*)

Broadcast either a Vector or arbitrary picklable data. If `data` is a Vector, the Vectors are resized and filled with the data from the Vector in the `root` node. If `data` is not a Vector, it is pickled, transmitted and returned from this function to all nodes.

Parameters

- **data** (*Vector* or any picklable object.) – The data to broadcast to the nodes.
- **root** (*int*) – The id of the node that is broadcasting the data.

Returns None (Vectors filled) or the transmitted data

Raises BroadcastError if `neuron.hoc.HocObjects` that aren't Vectors are transmitted

cell(*gid*, *nc*)

setup_transfer(**args*, ***kwargs*)

source_var(**args*, ***kwargs*)

target_var(**args*, ***kwargs*)

class patch.interpreter.**PythonHocInterpreter**

Bases: `object`

IClamp(*x=0.5*, *sec=None*)

NetCon(*source*, *target*, **args*, ***kwargs*)

NetStim(**args*, ***kwargs*)

ParallelCon(*a*, *b*, *output=True*, **args*, ***kwargs*)

ParallelContext()

PointProcess(**args*, ***kwargs*)

SEClamp(*sec*, *x=0.5*)

Section(**args*, ***kwargs*)

SectionRef(**args*, *sec=None*)

Segment(*interpreter*, *ptr*, *section*, ***kwargs*)

VecStim(*pattern=None*, **args*, ***kwargs*)

Vector(*interpreter*, *ptr*)

cas()

continuerun(*time_stop*, *add=False*)

finitialize(*initial=None*)

load_extension(*extension*)

nrn_load_dll(*path*)

property parallel

record(*target*)

classmethod register_hoc_object(*hoc_object_class*)

run()

property time

5.3 patch.objects module

class patch.objects.**Connectable**

Bases: object

class patch.objects.**IClamp**(*interpreter*, *ptr*)

Bases: *patch.objects.PythonHocObject*, *patch.objects.WrapsPointers*

class patch.objects.**NetCon**(*interpreter*, *ptr*)

Bases: *patch.objects.PythonHocObject*

record(*vector=None*)

class patch.objects.**NetStim**(**args*, ***kwargs*)

Bases: *patch.objects.PythonHocObject*, *patch.objects.Connectable*

class patch.objects.**PointProcess**(**args*, ***kwargs*)

Bases: *patch.objects.PythonHocObject*, *patch.objects.Connectable*, *patch.objects.WrapsPointers*

Wrapper for all point processes (membrane and synapse mechanisms).

stimulate(*pattern=None*, *weight=0.04*, *delay=0.0*, ***kwargs*)

Stimulate a point process.

Parameters

- **pattern** (*list[float]*) – Specific stimulus event times to play into the point process.
- **kwargs** – All keyword arguments will be passed set on the NetStim

class patch.objects.**PointerWrapper**(*attr*)

Bases: object

class patch.objects.**PythonHocObject**(*interpreter*, *ptr*)

Bases: object

__deref__(*obj*)

Magic method that is called when a strong reference needs to be removed from the object.

__neuron__()

Magic method that is called when this object is passed to NEURON.

`__ref__`(*obj*)

Magic method that is called when a strong reference needs to be stored on the object.

class `patch.objects.SEClamp`(*interpreter*, *ptr*)

Bases: *patch.objects.PythonHocObject*, *patch.objects.WrapsPointers*

class `patch.objects.Section`(**args*, ***kwargs*)

Bases: *patch.objects.PythonHocObject*, *patch.objects.Connectable*, *patch.objects.WrapsPointers*

`add_3d`(*points*, *diameters=None*)

Add new 3D points to this section xyz data.

Parameters

- **points** – A 2D array of xyz points.
- **diameters** (*float or array*) – A scalar or array of diameters corresponding to the points. Default value is the section diameter.

`connect`(*target*, **args*, ***kwargs*)

Connect this section to another one as child section.

`connect_points`(*target*, *x=None*, ***kwargs*)

Connect a Segment of this Section to a target. Usually used to connect the membrane potential to a point process.

`iclamp`(*x=0.5*, *delay=0*, *duration=100*, *amplitude=0*)

Create a current clamp on the section.

Parameters

- **x** (*float*) – Location along the segment from 0 to 1.
- **delay** (*float*) – Duration of the pre-step holding interval, from *0* to *delay* ms.
- **duration** (*float*) – Duration of the step interval, from *delay* to *delay* + *duration* ms.
- **amplitude** (*Union[float, List[float]]*) – Can be a single value to define the current during the step (*delay* to *delay* + *duration* ms), or a sequence to play after *delay* ms. This will play 1 value of the sequence into the clamp per timestep.

Returns The current clamp placed in the section.

Return type *objects.SEClamp*

`insert`(**args*, ***kwargs*)

Insert a mechanism into the Section.

property parent

Returns the parent of the Section, or `None`

property points

Return the 3d point information associated to this section.

`pop`()

Pop this section off the section stack.

`push`()

Return a context manager that pushes this Section onto the section stack and takes it off when the context is exited.

record(*x=None*)

Record the Section at a certain point.

Parameters **x** (*float*) – Arcpoint, defaults to `__arc__` if omitted.

set_dimensions(*length*, *diameter*)

Set the length and diameter of the piece of cable this Section will represent in the simulation.

set_segments(*segments*)

Set the number of discrete points where equations are solved during simulation.

synapse(*factory*, **args*, *store=False*, ***kwargs*)

Insert a synapse into the Section.

Parameters

- **factory** (*callable*) – Callable that creates a point process, is given the Section as first argument and passes on all other args.
- **store** (*bool*) – Store the synapse on the Section in a `synapses` attribute.

vclamp(*x=0.5*, *delay=0*, *duration=100*, *after=0*, *voltage=- 70*, *holding=- 70*)

Create a voltage clamp on the section.

Parameters

- **x** (*float*) – Location along the segment from 0 to 1.
- **delay** (*float*) – Duration of the pre-step holding interval, from *0* to *delay* ms.
- **duration** (*float*) – Duration of the step interval, from *delay* to *delay* + *duration* ms.
- **after** (*float*) – Duration of the post-step holding interval, from *delay* + *duration* to *delay* + *duration* + *after* ms.
- **voltage** (*Union[float, List[float]]*) – Can be a single value to define the voltage during the step (*delay* to *delay* + *duration* ms), or 3 values to define the pre-step, step and post-step voltages altogether.
- **holding** (*float*) – If *voltage* is a single value, *holding* is used for the pre-step and post-step voltages.

Returns The single electrode voltage clamp placed in the section.

Return type *objects.SEClamp*

wholetree()

Return the whole tree of child Sections

Return type List[patch.Section]

class patch.objects.**SectionRef**(*interpreter*, *ptr*)

Bases: *patch.objects.PythonHocObject*

property child

class patch.objects.**Segment**(*interpreter*, *ptr*, *section*, ***kwargs*)

Bases: *patch.objects.PythonHocObject*, *patch.objects.Connectable*, *patch.objects.WrapsPointers*

class patch.objects.**VecStim**(**args*, ***kwargs*)

Bases: *patch.objects.PythonHocObject*, *patch.objects.Connectable*

property pattern

property vector

class patch.objects.**Vector**(*interpreter*, *ptr*)

Bases: *patch.objects.PythonHocObject*

record(*target*, **args*, ***kwargs*)

class patch.objects.**WrapsPointers**

Bases: object

5.4 Module contents

CHAPTER

SIX

CANDY GUIDE

Read through this guide to get an idea of the sugar you're getting.

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

PYTHON MODULE INDEX

p

patch, 18
patch.core, 13
patch.interpreter, 14
patch.objects, 15

INDEX

Symbols

`__deref__()` (*patch.objects.PythonHocObject method*), 15

`__neuron__()` (*patch.objects.PythonHocObject method*), 15

`__ref__()` (*patch.objects.PythonHocObject method*), 15

A

`add_3d()` (*patch.objects.Section method*), 16

`assert_connectable()` (*in module patch.core*), 13

B

`broadcast()` (*patch.interpreter.ParallelContext method*), 14

C

`cas()` (*patch.interpreter.PythonHocInterpreter method*), 14

`cell()` (*patch.interpreter.ParallelContext method*), 14

`child` (*patch.objects.SectionRef property*), 17

`connect()` (*patch.objects.Section method*), 16

`connect_points()` (*patch.objects.Section method*), 16

`Connectable` (*class in patch.objects*), 15

`continuerun()` (*patch.interpreter.PythonHocInterpreter method*), 14

F

`finitialize()` (*patch.interpreter.PythonHocInterpreter method*), 14

I

`IClamp` (*class in patch.objects*), 15

`IClamp()` (*patch.interpreter.PythonHocInterpreter method*), 14

`iclamp()` (*patch.objects.Section method*), 16

`insert()` (*patch.objects.Section method*), 16

`is_point_process()` (*in module patch.core*), 13

`is_section()` (*in module patch.core*), 13

L

`load_extension()` (*patch.interpreter.PythonHocInterpreter method*), 14

M

`module`
- `patch`, 18
- `patch.core`, 13
- `patch.interpreter`, 14
- `patch.objects`, 15

N

`NetCon` (*class in patch.objects*), 15

`NetCon()` (*patch.interpreter.PythonHocInterpreter method*), 14

`NetStim` (*class in patch.objects*), 15

`NetStim()` (*patch.interpreter.PythonHocInterpreter method*), 14

`nrn_load_dll()` (*patch.interpreter.PythonHocInterpreter method*), 15

P

`parallel` (*patch.interpreter.PythonHocInterpreter property*), 15

`ParallelCon()` (*patch.interpreter.PythonHocInterpreter method*), 14

`ParallelContext` (*class in patch.interpreter*), 14

`ParallelContext()` (*patch.interpreter.PythonHocInterpreter method*), 14

`parent` (*patch.objects.Section property*), 16

`patch`
- `module`, 18

`patch.core`
- `module`, 13

`patch.interpreter`
- `module`, 14

`patch.objects`
- `module`, 15

`pattern` (*patch.objects.VecStim property*), 17

`PointerWrapper` (*class in patch.objects*), 15

`PointProcess` (*class in patch.objects*), 15

`PointProcess()` (*patch.interpreter.PythonHocInterpreter method*), 14

`points` (*patch.objects.Section property*), 16

`pop()` (*patch.objects.Section method*), 16

`push()` (*patch.objects.Section method*), 16

PythonHocInterpreter (*class in patch.interpreter*), 14
PythonHocObject (*class in patch.objects*), 15

R

record() (*patch.interpreter.PythonHocInterpreter method*), 15
record() (*patch.objects.NetCon method*), 15
record() (*patch.objects.Section method*), 16
record() (*patch.objects.Vector method*), 18
register_hoc_object() (*patch.interpreter.PythonHocInterpreter class method*), 15
run() (*patch.interpreter.PythonHocInterpreter method*), 15

S

SEClamp (*class in patch.objects*), 16
SEClamp() (*patch.interpreter.PythonHocInterpreter method*), 14
Section (*class in patch.objects*), 16
Section() (*patch.interpreter.PythonHocInterpreter method*), 14
SectionRef (*class in patch.objects*), 17
SectionRef() (*patch.interpreter.PythonHocInterpreter method*), 14
Segment (*class in patch.objects*), 17
Segment() (*patch.interpreter.PythonHocInterpreter method*), 14
set_dimensions() (*patch.objects.Section method*), 17
set_segments() (*patch.objects.Section method*), 17
setup_transfer() (*patch.interpreter.ParallelContext method*), 14
source_var() (*patch.interpreter.ParallelContext method*), 14
stimulate() (*patch.objects.PointProcess method*), 15
synapse() (*patch.objects.Section method*), 17

T

target_var() (*patch.interpreter.ParallelContext method*), 14
time (*patch.interpreter.PythonHocInterpreter property*), 15
transform() (*in module patch.core*), 13
transform_arc() (*in module patch.core*), 13
transform_netcon() (*in module patch.core*), 13
transform_record() (*in module patch.core*), 13

V

vclamp() (*patch.objects.Section method*), 17
VecStim (*class in patch.objects*), 17
VecStim() (*patch.interpreter.PythonHocInterpreter method*), 14
Vector (*class in patch.objects*), 18
vector (*patch.objects.VecStim property*), 18
Vector() (*patch.interpreter.PythonHocInterpreter method*), 14

W

wholetree() (*patch.objects.Section method*), 17
WrapsPointers (*class in patch.objects*), 18