NEAT

Release 0.9.1

unknown

Mar 25, 2021

CONTENTS

1 Structure 3

2 Audience 5

3 Python 7

4 Free software 9

5 History 11

6 Documentation 13

7 Indices and tables 157

Bibliography 159

Python Module Index 161

Index 163

NEAT is a python library for the study, simulation and simplification of morphological neuron models. NEAT accepts morphologies in the *de facto* standard .swc format [Cannon1998], and implements high-level tools to interact with and analyze the morphologies.

NEAT also allows for the convenient definition of morphological neuron models. These models can be simulated, through an interface with the NEURON simulator [Carnevale2004], or can be analyzed with two classical methods: *(i)* the separation of variables method [Major1993] to obtain impedance kernels as a superposition of exponentials and *(ii)* Koch's method to compute impedances with linearized ion channels analytically in the frequency domain [Koch1985]. Furthermore, NEAT implements the neural evaluation tree framework [Wybo2019] and an associated C++ simulator, to analyze subunit independence.

Finally, NEAT implements a new and powerful method to simplify morphological neuron models into compartmental models with few compartments [Wybo2020]. For these models, NEAT also provides a NEURON interface so that they can be simulated directly, and will soon also provide a NEST interface [Gewaltig2007].

CHAPTER

ONE

STRUCTURE

NEAT's main functionality is implemented throught a number of tree classes. *neat.STree* is the base class, implementing basic functionalities such as getting and iterating over nodes, as well as adding and removing nodes. Each class implements another layer of functionality over the class from which it inherits. Figure 1 provides an overview of the inheritance structure. For instance, *neat.MorphTree* inherits from *neat.STree*, and implements all functionality to load, store and interact with morphologies. *neat.PhysTree* then adds another layer of functionality by allowing the definition of electrical parameters.

NEAT furthermore has a number of other classes, notably to implement ion channels (*neat.IonChannel*) and to provide a high-level API for the simplification method described in [Wybo2021] (*neat.CompartmentFitter*).

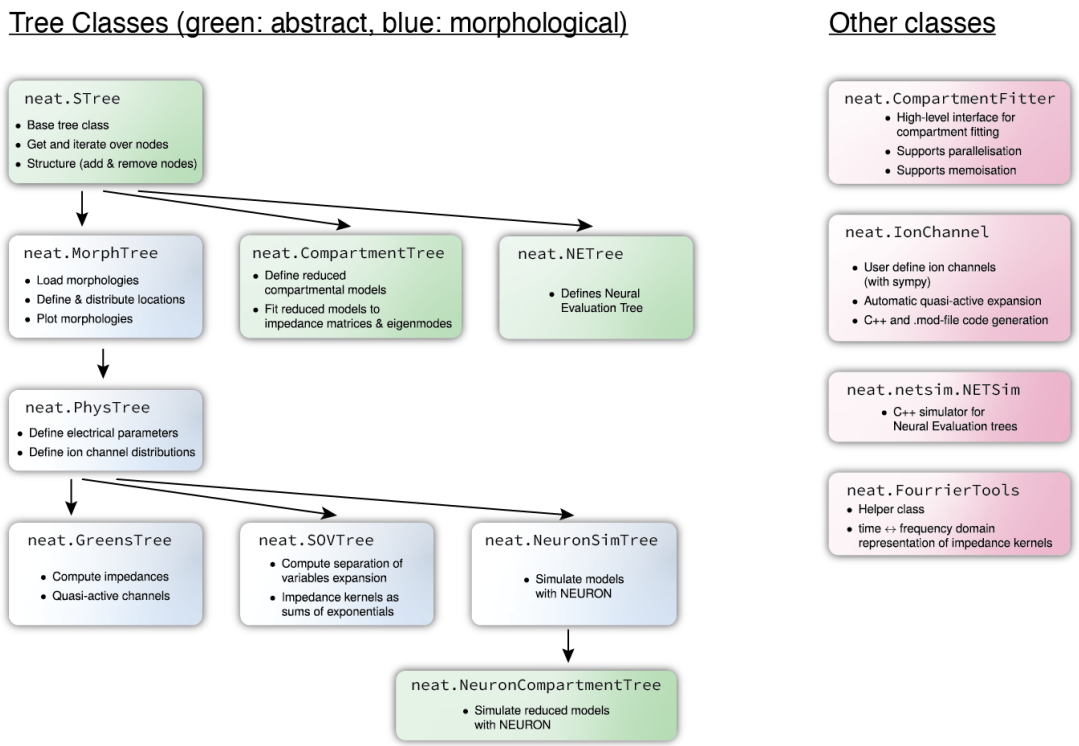


Fig. 1: **Figure 1.** Overview of NEAT structures. NEAT contains a number of tree classes, inheriting from *neat.STree*, as well as a number of helper classes.

A NEAT tree consists of nodes, and each tree class has a corresponding node class (Figure 2A). A tree class contains a *root* attribute (Figure 2B), which points to the corresponding node class instance that is the root of the tree (the soma, if the tree is a *MorphTree* or a derived class). Each node has an index (by which it can be accessed from the tree class),

a reference to its parent node (`None` if the node is the root), and a list containing references to its child nodes (empty if the node is a leaf).

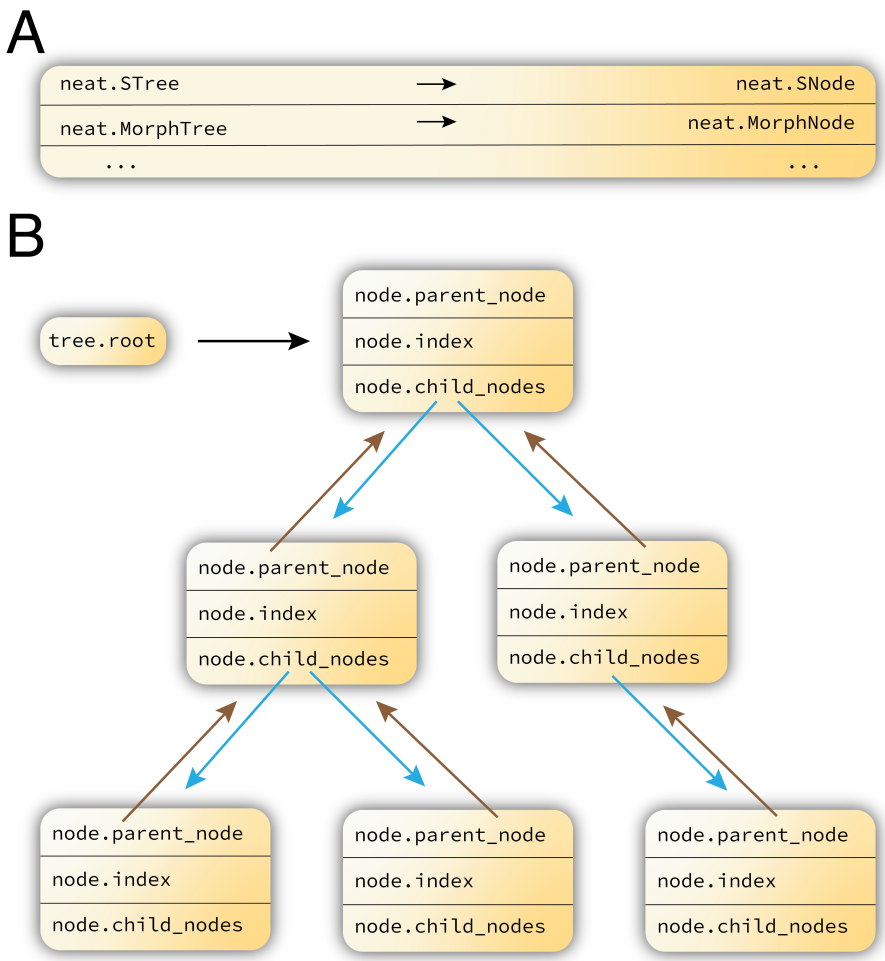


Fig. 2: **Figure 2.** Layout of a NEAT tree. **A:** Each NEAT tree consists of corresponding nodes. **B:** A tree contains a root node attribute, and each node has a parent node, an index and a list of child nodes.

CHAPTER

TWO

AUDIENCE

NEAT is of interest to neuroscientist who aim to understand dendritic computation, and to explore dendritic computation at the network level.

CHAPTER

THREE

PYTHON

Python is a powerful programming language that allows simple and flexible representations neural morphologies. Python has a vibrant and growing ecosystem of packages that NEAT uses to provide more features such as numerical linear algebra and drawing. In order to make the most out of NEAT you will want to know how to write basic programs in Python. Among the many guides to Python, we recommend the Python documentation and the text by Alex Martelli [Martelli03].

CHAPTER

FOUR

FREE SOFTWARE

NEAT is free software; you can redistribute it and/or modify it under the terms of the *GNU General Public License*. We welcome contributions. Join us on GitHub.

CHAPTER

FIVE

HISTORY

NEAT was born in April 2018. The original version was designed and written by Willem Wybo, based on code by Benjamin Torben-Nielsen. With help of Jakob Jordan and Benjamin Ellenberger, NEAT became an installable python package with documentation website.

Contributors are listed in *credits.*

CHAPTER

SIX

DOCUMENTATION

6.1 Installation

6.1.1 Install

Note: The following instructions are for Linux and Max OSX systems and only use command line tools. Please follow the appropriate manuals for Windows systems or tools with graphical interfaces.

You can install the latest release via pip:

```
pip install neatdend
```

The adventurous can install the most recent development version directly from our master branch (don't use this in production unless there are good reasons!):

```
git clone git@github.com:unibe-cns/NEAT.git
cd NEAT
pip install .
```

6.1.2 Post-Install

To use NEAT with NEURON, make sure NEURON is properly installed with its Python interface, and compile and install the default NEURON mechanisms by running

```
compilechannels default
```

To test the installation (requires *pytest*)

```
pytest
```

6.2 Tutorial

6.2.1 Interact with morphologies through `neat.MorphTree`

To illustrate some of the basic functionalities of `neat.MorphTree`, we'll read in and analyze a simple ball- and stick model.

[1]:
```
from neat import MorphTree, MorphNode
# load the model
m_tree = MorphTree(file_n='morph/ball_and_stick.swc')
```

A morphology consists of nodes, that each store the 3d coordinates and radii of the segments on the morphology. A node can be accessed by its index.

[2]:
```
# soma node (always index 1)
print(m_tree[1])
print(m_tree[1].xyz) # 3d coordinates
print(m_tree[1].R) # radius
# dendritic node
print(m_tree[4])
print(m_tree[4].xyz) # 3d coordinates
print(m_tree[4].R) # radius
```

```
SNode 1
[0. 0. 0.]
10.0
SNode 4
[499.   0.   0.]
1.0
```

Note that according to the `.swc` convention, the soma node has index 1. Morphologies should follow the three-point soma convention (http://neuromorpho.org/SomaFormat.html). Hence, the first dendritic node has index 4.

Iterating over nodes is intuitive and happens in a depth firs order. By default, nodes 2 and 3 are skipped in iterations.

[3]:
```
for node in m_tree: print(node)
```

```
SNode 1
SNode 4
SNode 5
```

We can also iterate over a subtree of the tree when we specify its root to the iterator.

[4]:
```
for node in m_tree.__iter__(m_tree[4]): print(node)
```

```
SNode 4
SNode 5
```

Treetype

Solving the cable equation on a morphology is often expensive, hence, `MorphTree` defines a "computational tree" that replaces the the individual nodes between bifurcations with equivalent cylinders. A second three of lower resolution is thus defined.

[5]:
```
m_tree.setCompTree()
```

Toggling between both trees is done by setting the `MorphTree.treetype` attribute:

[6]:
```
# let's print the nodes and their respective length and radii for both trees
m_tree.treetype = 'computational'
print('>>> computational tree')
for node in m_tree: print(str(node) + ', R (um) = ' + str(node.R) + ', L (um) = ' +␣
↪str(node.L))
m_tree.treetype = 'original'
```

(continues on next page)

(continued from previous page)

```
print('>>> original tree')
for node in m_tree: print(str(node) + ', R (um) = ' + str(node.R) + ', L (um) = ' +␣
↪str(node.L))
```

```
>>> computational tree
SNode 1, R (um) = 10.0, L (um) = 0.0
SNode 5, R (um) = 1.0, L (um) = 1001.0
>>> original tree
SNode 1, R (um) = 10.0, L (um) = 0.0
SNode 4, R (um) = 1.0, L (um) = 499.0
SNode 5, R (um) = 1.0, L (um) = 502.0
```

It is important to be aware of the fact that NEAT uses the concept of `treetype`, but if you only use NEAT functions, you will never have to set it yourself, as all functions automatically set the `treetype` they require.

Locations

Locations on a morphology are defined relative to the original node structure. A location consists of two coordinates: the node index and an x ($\in [0, 1]$)-coordinate, specifying the relative position of the location between the node and its parent node ($x = 0$ means that the location is at the parent node's 3d coordinates whereas $x = 1$ means that it's at the node's coordinates). Locations are coded as a tuple or a dictionary and collections of them can be stored under a given name.

[7]:
```
loc1 = {'node': 4, 'x': .2} # a location close to the soma
loc2 = (5, .8) # a location close to the dendritic tip
m_tree.storeLocs([loc1, loc2], name='my_favourite_locs')
```

Note that any NEAT function converts locations internally to :class:`MorphLoc` instances, that return the coordinates relative to the original or the computational tree based on which `treetype` is on. Locations can be retrieved as well (note that they are returned as :class:`MorphLoc` instances).

[8]:
```
locs = m_tree.getLocs('my_favourite_locs')
# returned as instances
print(locs)
# still print the original coordinates
print(locs[0])
# but returned coordinate value depends on `treetype`
m_tree.treetype = 'original'
print('original', locs[0]['node'], locs[0]['x'])
m_tree.treetype = 'computational'
print('computational', locs[0]['node'], locs[0]['x'])
```

```
[{'node': 4, 'x': 0.20 }, {'node': 5, 'x': 0.80 }]
{'node': 4, 'x': 0.20 }
original 4 0.2
computational 5 0.09970029970029971
```

Locations can also easily be compared

[9]:
```
locs[0] == (4, 0.2)
```

[9]:
```
1
```

A number of functions are implemented that distribute locations on the morphology (`MorphTree.distributeLocsOnNodes()`, `MorphTree.distributeLocsUniform()`, `MorphTree.distributeLocsRandom()`). We refer to the documentation for their specific uses. For now we'll distribute a set of locations uniformly on the morphology with a spacing of approximately 4 μm.

```
[10]: locs = m_tree.distributeLocsUniform(dx=4.)
```

We can also evaluate the path length (the length of the shortest path, in μm) between locations:

```
[11]: print(m_tree.pathLength(locs[0], locs[1]))
```

```
4.0040000000000004
```

Nodearg

To look at the more advanced functionality of `MorphTree`, we'll read in a real morphology

```
[12]: m_tree = MorphTree(file_n='morph/N19ttwt.CNG.swc')
m_tree.setCompTree()
```

In NEAT, functions that run over a set of nodes require a `node_arg`, which specifies which set of nodes. For all it's uses, we refer to the documentation of `MorphTree._convertNodeArgToNodes()`. Now, we'll use it to distribute locations uniformly on the subtree of node 76.

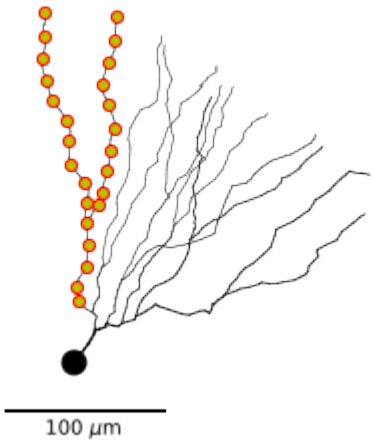
```
[13]: subtree_locs = m_tree.distributeLocsUniform(dx=15., node_arg=m_tree[74], name='subtree
↪')
```

Note that this set of locations is stored under the name `'subtree'`

Plotting

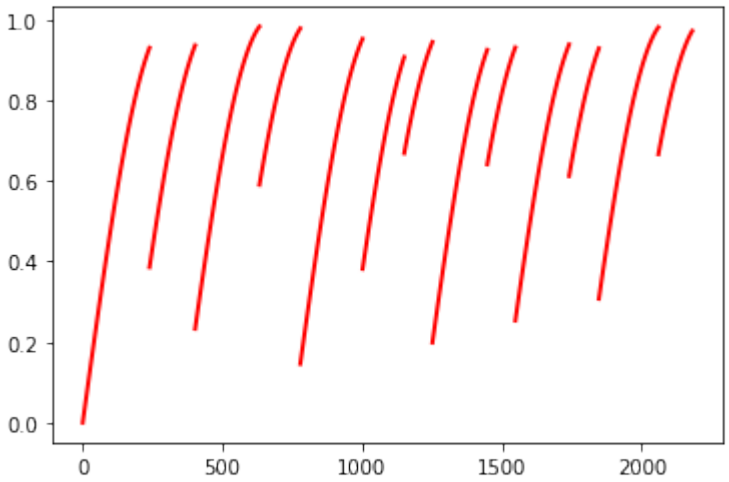
We can plot 2D projections of the morphology with `MorphTree.plot2DMorphology()`. This function has many options, we'll use this function to plot the morphology of the granule cell in combination with the locations defined before.

```
[14]: import matplotlib.pyplot as pl
pl.figure()
ax = pl.gca() # axes object needs to be passed to NEAT plot function
m_tree.plot2DMorphology(ax, marklocs=subtree_locs, locargs=dict(marker='o', mec='r',␣
↪mfc='y'))
pl.show()
```



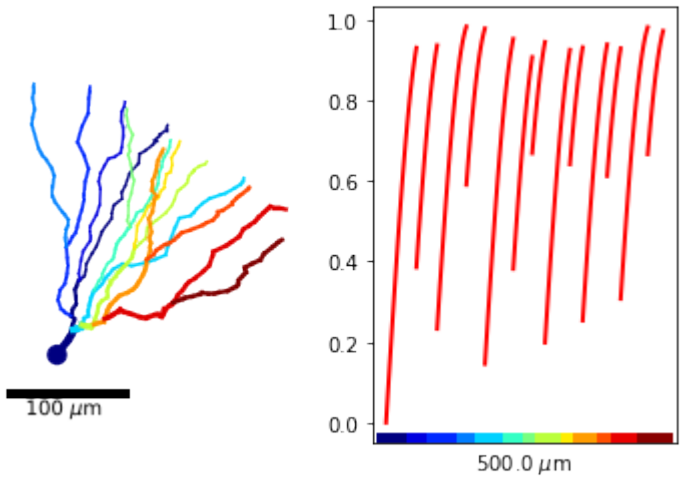
We can also unravel the morphology on a 1 dimensional axis (depth-first ordering). That way we can visualize the value of field defined on the morphology (such as membrance voltage).

[15]:
```
# We have to define a set of locations on the morphology at which the function is␣
↪evaluated (here dx ~ 4 um)
m_tree.makeXAxis(4.)
# create a function
import numpy as np
d2s = m_tree.distancesToSoma('xaxis') # the distances to the soma (in um) of the␣
↪locations stored under 'xaxis'
y_arr = np.sin(d2s/200.) # function to be plotted
# plot the function
ax = pl.figure().gca()
m_tree.plot1D(ax, y_arr, lw='2', c='r') # we can use the same keyword arguments as␣
↪for the matplotlib plot function
pl.show()
```



As such, it is not very clear which x-value corresponds with which location on the morphology. To remedy this, we can color each branch of the morphology differently and color the x-axis accordingly.

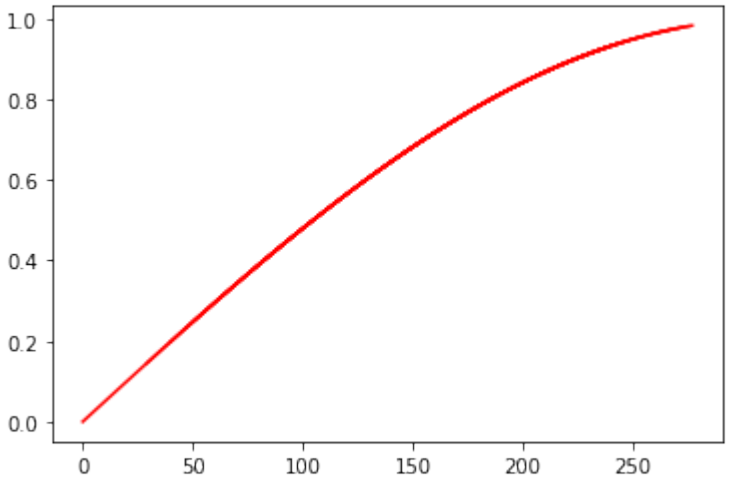
[16]:
```
from matplotlib import cm
pl.figure()
ax1 = pl.subplot(121); ax2 = pl.subplot(122)
# to color the x-axis, node colors have to be set first
m_tree.setNodeColors()
# plot the morphology with associated colors
m_tree.plot2DMorphology(ax1, cs='x_color', cmap=pl.get_cmap('jet'), plotargs={'lw':3})
# plot the function
m_tree.plot1D(ax2, y_arr, lw='2', c='r')
# color the x-axis of the plot according to the branch where each location is situated
m_tree.colorXAxis(ax2, cmap=pl.get_cmap('jet'))
```



Note that `MorphTree.colorXAxis()` gets its y-coordinates for the colored line as the axes limits. If the limits are changed after the colored axis is added, this may result in funky position for the colored line. Hence, only call this function after all lines are added to the plot.

We can also plot fields defined on the morphology as a function of the distance to the soma.

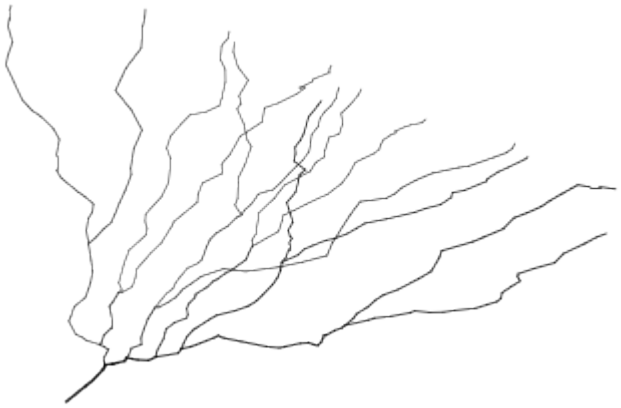
[17]:
```
ax = pl.figure().gca()
m_tree.plotTrueD2S(ax, y_arr, c='r')
pl.show()
```



Note that here we only see a single line as our field is only a function of the distance to the soma.

Finally, we may want to select a custom set of locations on the morphology. To get the node indices, NEAT implements an interactive plot function that, when one clicks on a location of the morphology, the indices of nearby nodes are printed as well as their distances to the soma (although the interactive part does not seem to run in the notebook, but it works when you just run python).

[18]:
```
m_tree.plotMorphologyInteractive()
```



Resampling

Some applications may require a resampling of the tree. A tree can be resampled starting from any given set of locations (although low resolution sets may give funny results). The positions of the nodes of the new tree in 3d space will then correspond to positions of the locations on the old tree. Note that the soma or the original tree is added by default, even if it is not in the set of locations.

[19]:
```
# create the resampled tree
m_tree_resampled = m_tree.createNewTree('subtree')
# plot the resampled tree
ax = pl.figure().gca()
m_tree_resampled.plot2DMorphology(ax)
pl.show()
```

6.2.2 Define morphological models with `neat.PhysTree`

The Class `neat.PhysTree` is used to define the physiological parameters of neuron models. It inherits from `neat.MorphTree` and thus has all its functionality. Just as `neat.MorphTree`, instances are initialized based on the standard `.swc` format:

[1]:
```
from neat import PhysTree
ph_tree = PhysTree(file_n='morph/L23PyrBranco.swc')
```

Defining physiological parameters

A `PhysTree` consists of `neat.PhysNode` instances, which inherit from `neat.MorphNode`. Compared to a `MorphNode`, a `PhysNode` has extra attributes (initialized to some default value) defining physiological parameters:

[2]:
```
# specific membrance capacitance (uF/cm^2)
print('default c_m node 1:', ph_tree[1].c_m)
# axial resisitance (MOhm*cm)
print('default r_a node 1:', ph_tree[1].r_a)
# point-like shunt located at {'node': node.index, 'x': 1.} (uS)
print('default g_shunt node 1:', ph_tree[1].g_shunt)
# leak and ion channel currents, stored in a dict with
# key: 'channel_name', value: [g_max, e_rev]
print('default currents node 1:', ph_tree[1].currents)
```

```
default c_m node 1: 1.0
default r_a node 1: 0.0001
default g_shunt node 1: 0.0
default currents node 1: {}
```

It is not recommended, and for ion channels even forbidden, to set the parameters directly via the nodes. Rather, the parameters should be specified with associated functions of `PhysTree`. These functions accept a `node_arg` keyword argument, which allows selecting a specific set of nodes. Parameters, can be given as `float`, in which case all nodes in `node_arg` will be set to the same value, a `dict` of `{node.index: parameter_value}`, or a callable function where the input is the distance of the middle of the node (`loc = {'node': node.index, 'x': .5}`) to the soma and the output the parameter.

Let's use `PhysTree.setPhysiology()` to set capacitance (1st argument) and axial resistance (2nd argument) in the whole tree:

[3]:
```
ph_tree.setPhysiology(lambda x: .8 if x < 60. else 1.6*.8, 100.*1e-6)
```

Here, we defined the capacitance to be .8 μF/cm^2 when the mid-point of the node is less than 60 μm from the soma and $1.6 * .8$ μF/cm^2, a common factor to take dendritic spines into account. Axial resistance was set to a constant value throughout the tree.

To set ion channels (see the 'Ionchannels in NEAT' tutorial on how to create your own ion channels), we must create the ion channel instance first. Here, we'll set a default sodium and potassium channel:

[4]:
```
from neat.channels.channelcollection.channelcollection import Na_Ta, Kv3_1
# create the ion channel instances
na_chan = Na_Ta()
k_chan = Kv3_1()
# set the sodium channel only at the soma, with a reversal of 50 mV
ph_tree.addCurrent(na_chan, 1.71*1e6, 50., node_arg=[ph_tree[1]])
# set the potassium channel throughout the dendritic tree, at 1/10th
# of its somatic conductance, and with a reversal of -85 mV
```

(continues on next page)

(continued from previous page)

```
gk_soma = 0.45*1e6
ph_tree.addCurrent(k_chan, lambda x: gk_soma if x < .1 else gk_soma/10., -85.)
```

Now, we only have to set the leak current. We have two possibilities for this: (i) we could set the leak current by providing conductance and reversal in the standard way with `PhysTree.setLeakCurrent()` or (ii) with could fit the leak current to fix equilibrium potential and membrane time scale (if possible) with `PhysTree.fitLeakCurrent()`. We take the second option here:

[5]:
```
# fit leak current to yield an equilibrium potential of -70 mV and
# a total membrane time-scale of 10 ms (with channel opening
# probabilities evaluated at -70 mV)
ph_tree.fitLeakCurrent(-70., 10.)
```

Inspecting the physiological parameters

We can now inspect the contents of various `PhysNode` instances:

[6]:
```
# soma node
print(ph_tree[1])
```

```
SNode 1 --- (r_a = 9.999999999999999e-05 MOhm*cm, g_Na_Ta = 1710000.0 uS/cm^2, g_Kv3_
↪1 = 450000.0 uS/cm^2, g_L = 31.540810731846726 uS/cm^2, c_m = 0.8 uF/cm^2)
```

[7]:
```
# dendrite node
print(ph_tree[115])
```

```
SNode 115, Parent: SNode 115 --- (r_a = 9.999999999999999e-05 MOhm*cm, g_Kv3_1 =␣
↪45000.0 uS/cm^2, g_L = 123.19344287327341 uS/cm^2, c_m = 1.2800000000000002 uF/cm^2)
```

Or, to get the full information on conductances and reversal potentials of membrane currents:

[8]:
```
# soma node
print(ph_tree[1].currents)
```

```
{'Na_Ta': [1710000.0, 50.0], 'Kv3_1': [450000.0, -85.0], 'L': [31.540810731846726, -
↪48.63880498727144]}
```

[9]:
```
# dendrite node
print(ph_tree[115].currents)
```

```
{'Kv3_1': [45000.0, -85.0], 'L': [123.19344287327341, -69.41475491536457]}
```

Active dendrites compared to closest passive version

Imagine we aim to investigate the role of active dendritic channels, and to that purpose want to compare the active dendritic tree with a passive version. We may compute the leak conductance of this "passified" tree as the sum of all ion channel conductance evaluate at the equilibrium potential. The equilibrium potentials is stored on the tree using `PhysTree.setEEq()`:

[10]:
```
ph_tree.setEEq(-70.)
```

To obtain the passified tree, we use `PhysTree.asPassiveMembrane()`. However, this function will overwrite the parameters of the original nodes, if we want to maintain the initial tree, we have to copy it first:

[11]:
```
# copy the initial tree
ph_tree_pas = ph_tree.__copy__()
# set to passive (except the soma)
ph_tree_pas.asPassiveMembrane([n for n in ph_tree_pas if n.index != 1])
```

We can now inspect the nodes:

[12]:
```
# soma node
print(ph_tree_pas[1])
```

```
SNode 1 --- (r_a = 9.999999999999999e-05 MOhm*cm, g_Na_Ta = 1710000.0 uS/cm^2, g_Kv3_
↪1 = 450000.0 uS/cm^2, g_L = 31.540810731846726 uS/cm^2, c_m = 0.8 uF/cm^2)
```

[13]:
```
# dendrite node
print(ph_tree_pas[115])
```

```
SNode 115, Parent: SNode 115 --- (r_a = 9.999999999999999e-05 MOhm*cm, g_L = 127.
↪99999999999991 uS/cm^2, c_m = 1.2800000000000002 uF/cm^2)
```

And the currents:

[14]:
```
# soma node
print(ph_tree_pas[1].currents)
```

```
{'Na_Ta': [1710000.0, 50.0], 'Kv3_1': [450000.0, -85.0], 'L': [31.540810731846726, -
↪48.63880498727144]}
```

[15]:
```
# dendrite node
print(ph_tree_pas[115].currents)
```

```
{'L': [127.99999999999991, -70.0]}
```

Comparing this to the previously shown nodes of the full tree, we see that the dendrite nodes have been "passified".

Computational tree

The computational tree in `PhysTree` works the same as in `MorphTree`, except that it's derivation also considers changes in physiological parameters, next to changes in morphological parameters.

[16]:
```
ph_tree.setCompTree()
# compare number of nodes in computational tree and original tree
ph_tree.treetype = 'original'
print('%d nodes in original tree'%(len(ph_tree)))
ph_tree.treetype = 'computational'
print('%d nodes in computational tree'%(len(ph_tree)))
```

```
432 nodes in original tree
98 nodes in computational tree
```

Compare this to the number of nodes in the computational tree induced solely by the morphological parameters:

[17]:
```
from neat import MorphTree
m_tree = MorphTree('morph/L23PyrBranco.swc')
m_tree.setCompTree()
m_tree.treetype = 'computational'
print('%d nodes in computational `MorphTree`'%len(m_tree))
```

```
87 nodes in computational `MorphTree`
```

Note: only call this `PhysTree.setCompTree` when all physiological parameters have been set, and ***never*** change parameters stored at individual nodes when `treetype` is `computational`, as this leads to the computational tree being inconsistent with the original tree.

6.2.3 Simulate models with `neat.NeuronSimTree`

NEAT implements an interface to the NEURON simulator, so that models defined by neat-trees can be simulated with the NEURON simulator. To have access to this functionality, the NEURON simulator and it's Python interface need to be installed.

The class `neat.NeuronSimTree` implements this interface, and inherits from `neat.PhysTree`. Hence, a `NeuronSimTree` can be defined in the same way as a `PhysTree`.

[18]:
```
from neat import NeuronSimTree
sim_tree = NeuronSimTree(file_n='morph/L23PyrBranco.swc')
sim_tree.setPhysiology(lambda x: .8 if x < 60. else 1.6*.8, 100.*1e-6)
# ... etc
```

The `__copy__` function

If a different type of tree is needed than the one originality defined, a handy feature of NEAT's copy function can be used: we can specify the type of tree we want as a keyword argument to `neat.MorphTree.__copy__()`. This function then copies all attributes that both tree classes have in common. Since `NeuronSimTree` is a subclass of `PhysTree`, we end up with an identical tree, but with additional functions and associated attributes to simulate the associated NEURON model.

[19]:
```
sim_tree = ph_tree.__copy__(new_tree=NeuronSimTree())
```

Setting up a simulation

First, we must initialize the tree structure into hoc.

[20]:
```
sim_tree.initModel(t_calibrate=100.)
```

We may then add inputs to the tree. NEAT implements a number of standard synapse types and current injections. Let's apply a DC current step to the soma and also give some input to a conductance-based dendritic synapse.

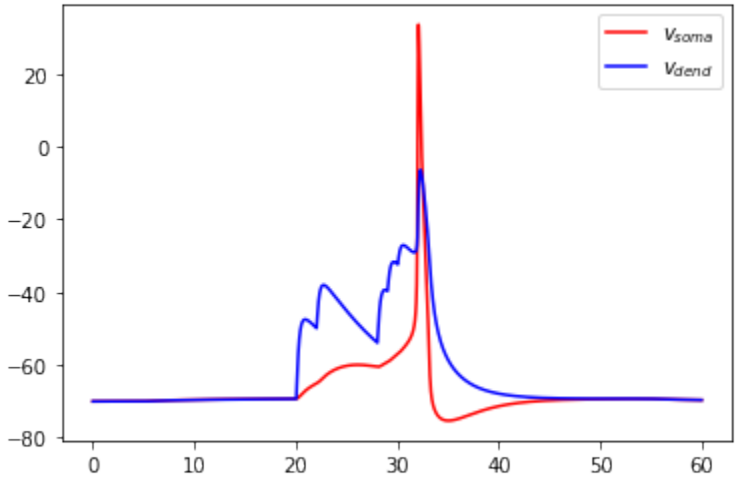
[21]:
```
# somatic DC current step with amplitude = 0.100 nA, delay = 5. ms and duration = 50.␣
↪ms
sim_tree.addIClamp((1.,.5), 0.010, 5., 50.)
# dendritic synapse with rise resp. decay times of .2 resp 3. ms and reversal of 0 mV
sim_tree.addDoubleExpSynapse((115,.8), .2, 3., 0.)
# give the dendritic synapse a weight of 0.005 uS and connect it to an input spike␣
↪train
sim_tree.setSpikeTrain(0, 0.005, [20.,22.,28.,29.,30.])
```

We will record voltage at the somatic and dendritic site. Recording locations should be stored under the name 'rec locs'.

[22]:
```
sim_tree.storeLocs([(1,.5), (115,.8)], name='rec locs')
```

We can then run the model for 60 ms and plot the results:

[23]:
```
# simulate the model and delete all hoc-variables afterwards
res = sim_tree.run(60.)
sim_tree.deleteModel()

# plot the results
import matplotlib.pyplot as pl
pl.plot(res['t'], res['v_m'][0], c='r', label=r'$v_{soma}$')
pl.plot(res['t'], res['v_m'][1], c='b', label=r'$v_{dend}$')
pl.legend(loc=0)
pl.show()
```



User defined point-process

Note that it is very easy to add user defined point-process to the `NeuronSimTree`. In fact, all any of the default functions to add point-process do, is defining a `neat.MorphLoc` based on the input, so that the point process is added at the right coordinates no matter whether `treetype` was 'original' or 'computational'. All hoc sections are stored in the dict `self.sections` which has as keys the node indices. Hence, in pseudo code one would do:

```
loc = neat.MorphLoc((node.index, x-coordinate), sim_tree)

# define the point process at the correct location
pp = h.user_defined_point_process(sim_tree.sections[loc['node']](loc['x']))

# set its parameters
pp.param1 = val1
pp.param2 = val2
...

# store the point process (e.g. if it is a synapse in `sim_tree.syns`)
sim_tree.syns.append(pp)
```

6.2.4 Evaluate impedance matrices with `neat.GreensTree`

The class `neat.GreensTree` inherits from `neat.PhysTree` and implements Koch's algorithm [-@Koch1984] to calculate impedances in the Fourrier domain. For a given input current of frequency ω at location x, the impedance gives the linearized voltage response at a location x':

$$v_{x'}(\omega) = z_{x'x}(\omega)\, i_x(\omega). \tag{6.1}$$

Applying the inverse Fourrier transform yields a convolution in the the time domain:

$$v_{x'}(t) = z_{x'x}(t) * i_x(t), \tag{6.2}$$

with we call $z_{x'x}(t) = FT^{-1}\left(z_{x'x}(\omega)\right)$ the impedance kernel. The steady state impedance is then:

$$z_{x'x} = \int_0^\infty \mathrm{d}t\, z_{x'x}(t) = z_{x'x}(\omega = 0). \tag{6.3}$$

Computing an impedance kernel

To compute an impedance kernel, we first have to initialize the `GreensTree`:

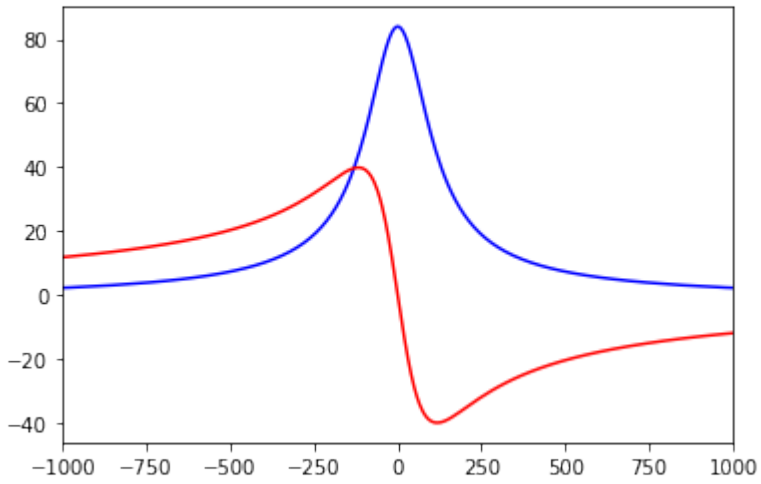
[24]:
```
from neat import GreensTree
greens_tree = ph_tree.__copy__(new_tree=GreensTree())
```

For the calculation to proceed efficiently, `GreensTree` first sets effective, frequency-dependent boundary conditions for each cylindrical section. Hence we must specify the frequencies at which we want to evaluate impedances. If we aim to also compute temporal kernels, `neat.FourrierTools` is a handy tool to obtain the correct frequencies. Suppose for instance that we aim to compute an impedance kernels from 0 to 50 ms:

[25]:
```
from neat import FourrierTools
import numpy as np
# create a Fourriertools instance with the temporal array on which to evaluate the␣
↪impedance kernel
t_arr = np.linspace(0.,50.,1000)
ft = FourrierTools(t_arr)
# appropriate frequencies are stored in `ft.s`
# set the boundary condition for cylindrical segments in `greens_tree`
greens_tree.setImpedance(ft.s)
```

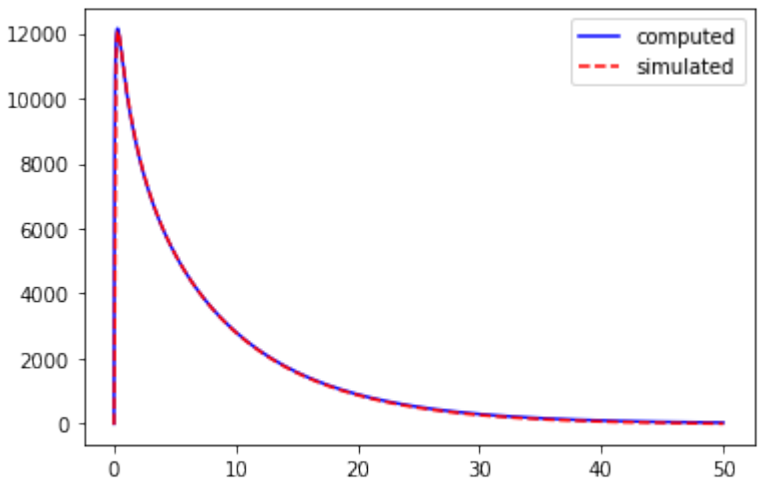
We can now compute for instance the impedance kernel between dendritic and somatic site:

[26]:
```
z_trans = greens_tree.calcZF((1,.5), (115,.8))

# plot the kernel
pl.plot(ft.s.imag, z_trans.real, 'b')
pl.plot(ft.s.imag, z_trans.imag, 'r')
pl.xlim((-1000.,1000.))
pl.show()
```



We can also obtain this kernel in the time domain with the `FourrierTools` object:

[27]:
```
# time domain kernel
tt, zt = ft.ftInv(z_trans)

# comparison with NEURON simulation
sim_tree.initModel(t_calibrate=300.)
i_amp, i_dur = 0.001, 0.1
sim_tree.addIClamp((115,.8), i_amp, 0., i_dur)
res = sim_tree.run(50.)
sim_tree.deleteModel()
res['v_m'] -= res['v_m'][:,-1]
res['v_m'] /= (i_amp*1e-3*i_dur)

# plot the kernel
pl.plot(tt, zt.real, 'b', label='computed')
pl.plot(res['t'], res['v_m'][0], 'r--', label='simulated')
pl.legend(loc=0)
pl.show()
```



Computing the impedance matrix

While `GreensTree.calcZF()` could be used to explicitely compute the impedance matrix, `GreensTree.calcImpedanceMatrix()` uses the symmetry and transitivity properties of impedance kernels to further optimize the calculation.

[28]:
```
z_locs = [(1,.5), (115,.8)]
z_mat = greens_tree.calcImpedanceMatrix(z_locs)
```

This matrix has shape `(len(ft.s), len(z_locs), len(z_locs))`. The zero-frequency component is at `ft.ind_0s`. Hence, the following gives the steady state impedance matrix:

[29]:
```
print(z_mat[ft.ind_0s])
```

```
[[ 88.07091747+0.j  83.93186356+0.j]
 [ 83.93186356+0.j 183.59769195+0.j]]
```

6.2.5 Simplify a model with `neat.CompartmentFitter`

The class `neat.CompartmentFitter` is used to obtain simplified compartmental models, where the parameters of the compartments are optimized to reproduce voltages at any set of locations on the morphology. It is initialized based on a `neat.PhysTree`:

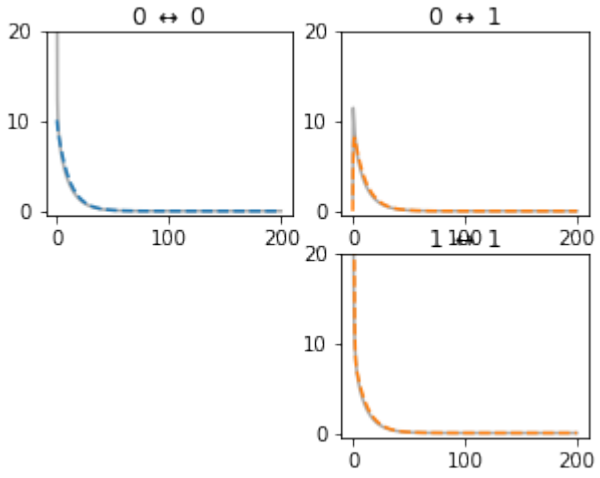
[30]:
```
from neat import CompartmentFitter
c_fit = CompartmentFitter(ph_tree)
```

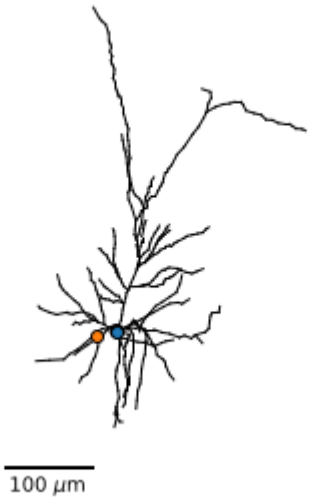
The function `CompartmentFitter.fitModel()` then returns a `neat.CompartmentTree` object defining the simplified model, with the parameters of the optimized compartments:

[31]:
```
# compute a simplified tree containing a somatic and dendritic compartment
f_locs = [(1,.5), (115,.8)]
c_tree = c_fit.fitModel(f_locs, use_all_channels_for_passive=False)
```

One way to understand whether the reduction will be faithful, is to check whether the passive reduced model reproduces the same impedance kernels as the full model. The `checkPassive()` function of `neat.CompartmentFitter` allows the comparison of these kernels between full and reduced models:

[32]:
```
c_fit.checkPassive(f_locs, use_all_channels_for_passive=False)
```





The simplified model `neat.CompartmentTree`

Each `neat.CompartmentNode` in the `CompartmentTree` stores the optimized parameters of the compartment, and the coupling conductance with it's parent node.

[33]:
```
print(c_tree)
```

```
>>> Tree
    SNode 0 --- (g_c = 0.000000000000 uS, g_L = 0.009188301476 uS, g_Na_Ta = 14.
↪378466892362 uS, g_Kv3_1 = 7.000477231569 uS, c = 0.000099142300 uF)
    SNode 1, Parent: SNode 1 --- (g_c = 0.009224434868 uS, g_L = 0.000411034483 uS, g_
↪Na_Ta = 0.000000000000 uS, g_Kv3_1 = 0.072296414995 uS, c = 0.000004268327 uF)
```

To keep track of the mapping between compartments and locations, each `CompartmentNode` also has a `loc_ind` attribute, containing the index of the location in the original list given to `CompartmentFitter.fitModel()` (here `f_locs`) to which the compartment is fitted:

[34]:
```
# node 0 corresponds to location 0 in `f_locs`
print('node index: %d, loc index: %d'%(c_tree[0].index, c_tree[0].loc_ind))
# node 1 corresponds to location 1 in `f_locs`
print('node index: %d, loc index: %d'%(c_tree[1].index, c_tree[1].loc_ind))
```

```
node index: 0, loc index: 0
node index: 1, loc index: 1
```

Here, these indices correspond, but in general there is ***no*** guarantee this will be the case. A list of 'fake' locations for the compartmental model can also be obtained. These locations contain nothing but the node index and an x-coordinate without meaning.

[35]:
```
c_locs = c_tree.getEquivalentLocs()
print(c_locs[0], c_locs[1])
```

```
(0, 0.5) (1, 0.5)
```

Simulate the simplified model with `neat.NeuronCompartmentTree`

The simplified model can be simulated directly in NEURON, with the same API as `neat.NeuronSimTree`. To do so, the function `neat.createReducedNeuronModel()` converts the `CompartmentTree` to a `neat.NeuronCompartmentTree`:

[36]:
```
from neat import createReducedNeuronModel
c_sim_tree = createReducedNeuronModel(c_tree)
```

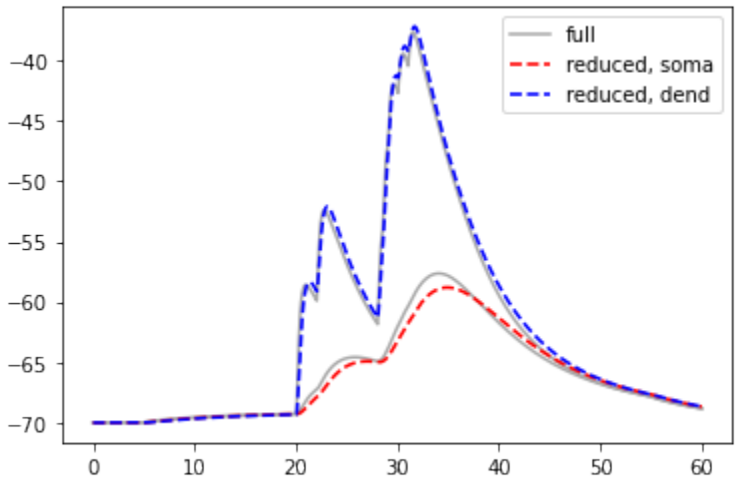
We may now check whether our simplification was successful by running the same simulation for the full and reduced models:

[37]:
```
# initialize, run and delete the full model, set input locations as stored in `f_locs`
sim_tree.initModel(t_calibrate=100.)
sim_tree.addIClamp(f_locs[0], .01, 5., 50.)
sim_tree.addDoubleExpSynapse(f_locs[1], .2, 3., 0.)
sim_tree.setSpikeTrain(0, 0.002, [20.,22.,28.,28.5,29.,30.,31.])
sim_tree.storeLocs(f_locs, name='rec locs')
res_full = sim_tree.run(60.)
sim_tree.deleteModel()

# initialize and run the simplified model, set input locations as stored in `c_locs`
c_sim_tree.initModel(t_calibrate=100.)
c_sim_tree.addIClamp(c_locs[0], 0.01, 5., 50.)
c_sim_tree.addDoubleExpSynapse(c_locs[1], .2, 3., 0.)
c_sim_tree.setSpikeTrain(0, 0.002, [20.,22.,28.,28.5,29.,30.,31.])
c_sim_tree.storeLocs(c_locs, name='rec locs')
res_reduced = c_sim_tree.run(60.)
# print the hoc topology of the reduced model
from neuron import h
h.topology()
# delete the reduced model
c_sim_tree.deleteModel()
```

```
|-|       0(0-1)
  `|       1(0-1)
```

We compare somatic and dendritic voltages in both models:

[38]:
```
pl.plot(res_full['t'], res_full['v_m'][0], c='DarkGrey', label='full')
pl.plot(res_full['t'], res_full['v_m'][1], c='DarkGrey')
pl.plot(res_reduced['t'], res_reduced['v_m'][0], 'r--', lw=1.6, label='reduced, soma')
pl.plot(res_reduced['t'], res_reduced['v_m'][1], 'b--', lw=1.6, label='reduced, dend')
pl.legend(loc=0)
pl.show()
```



6.2.6 Compute the separation of variables expansion with `neat.SOVTree`

The separation of variables expansion [-@Major1993] computes impedance kernels as superpositions of exponentials:

$$z_{x'x}(t) = \sum_{k=0}^{\infty} e^{-\frac{t}{\tau_k}} \phi_k(x') \phi_k(x) \tag{6.4}$$

which can be used to compute PSP responses to current inputs analytically, or even to simulate the neuron model as:

$$\dot{y}_k(t) = -\frac{y_k(t)}{\tau_k} + \int \mathrm{d}x \, \phi_k(x) \, I(x, t, V(x,t)) \quad \forall k \tag{6.5}$$

$$V(x,t) = \sum_k \phi_k(x) \, y_k(t), \tag{6.6}$$

with $I(x, t, V(x,t))$ the possibly voltage-dependent input current at location x along the dendrite and at time t.

`neat.SOVTree` implements Major's algorithm [-@Major1993] to compute the SOV solution. A `neat.SOVTree` is initialized like any other `neat` tree:

[1]:
```
import numpy as np
import matplotlib.pyplot as pl

from neat import SOVTree

# load the tree
sov_tree = SOVTree(file_n='morph/N19ttwt.CNG.swc')
# capacitane 1 uF/cm^2 and axial resistance 0.0001 MOhm*cm
sov_tree.setPhysiology(1., 0.0001)
# fit the leak conductance to have an equilibrium potential of -75. and a
# uniform membrane time scale of 10 ms
sov_tree.fitLeakCurrent(-75., 10.)
# set the computational tree
sov_tree.setCompTree()
```

Then, the exponential time-scales τ_k are computed as the zeros of the transcendental equation.

```
[2]: # construct transcendental equation
     sov_tree.calcSOVEquations(maxspace_freq=100.)
```

We may then obtain the exponential time-scales τ_k and associated spatial functions $\phi_k(x_i)$, the latter evaluated at a set of locations $(x_1, \ldots, x_N)$. In this case, we will only plot the spatial modes, so we define a set of locations that is good for plotting spatial functions defined on the morphology ('xaxis').

```
[3]: # creates and stores (under the name 'xaxis') a set of locations on the morphology␣
     ↪good for plotting
     sov_tree.makeXAxis()
```

To asses the importance of a mode for the voltage dynamics at a given set of locations, we use the metric

$$\mathrm{Imp}_k = \sqrt{\tau_k \left(\sum_{i=0}^{N} \sum_{j=0}^{N} \phi_k(x_i)\, \phi_k(x_j) \right)} \tag{6.7}$$

We may then obtain the most important SOV terms at the set of locations stored under 'xaxis':

```
[4]: alphas, phimat = sov_tree.getSOVMatrices('xaxis')
```

We can also define a cutoff threshold. Modes with relative importance below this threshold ($\mathrm{Imp}_k < \epsilon\, \mathrm{Imp}_0$) are not returned

```
[5]: alphas_, phimat_ = sov_tree.getImportantModes(locarg='xaxis', eps=1e-3)
     # compare the size of alphas with the size of alphas_
     print('Number of SOV terms in alphas =', alphas.shape[0])
     print('Number of SOV terms in alphas_ =', alphas_.shape[0])
```

```
Number of SOV terms in alphas = 120
Number of SOV terms in alphas_ = 37
```

Note furthermore that this function returns the reciprocals of the timescales $\alpha_k = \frac{1}{\tau_k}$, in kHz. The time-scales can be obtained in ms as:

```
[6]: print('tau_k (k=0,...36) =\n',1./alphas_)
```

```
tau_k (k=0,...36) =
 [10.          1.17930791  0.87175063  0.74621091  0.61592058  0.55352705
  0.51001138  0.46019773  0.33093398  0.22778449  0.18268101  0.1810243
  0.15825737  0.1555517   0.09367747  0.08063473  0.07379258  0.06509325
  0.05731892  0.0533252   0.04952842  0.04547815  0.03806818  0.03576512
  0.02960574  0.0277235   0.02652117  0.02224916  0.02059959  0.02033902
  0.02004149  0.0193636   0.01900246  0.01658901  0.01623951  0.01467919
  0.01362795]
```

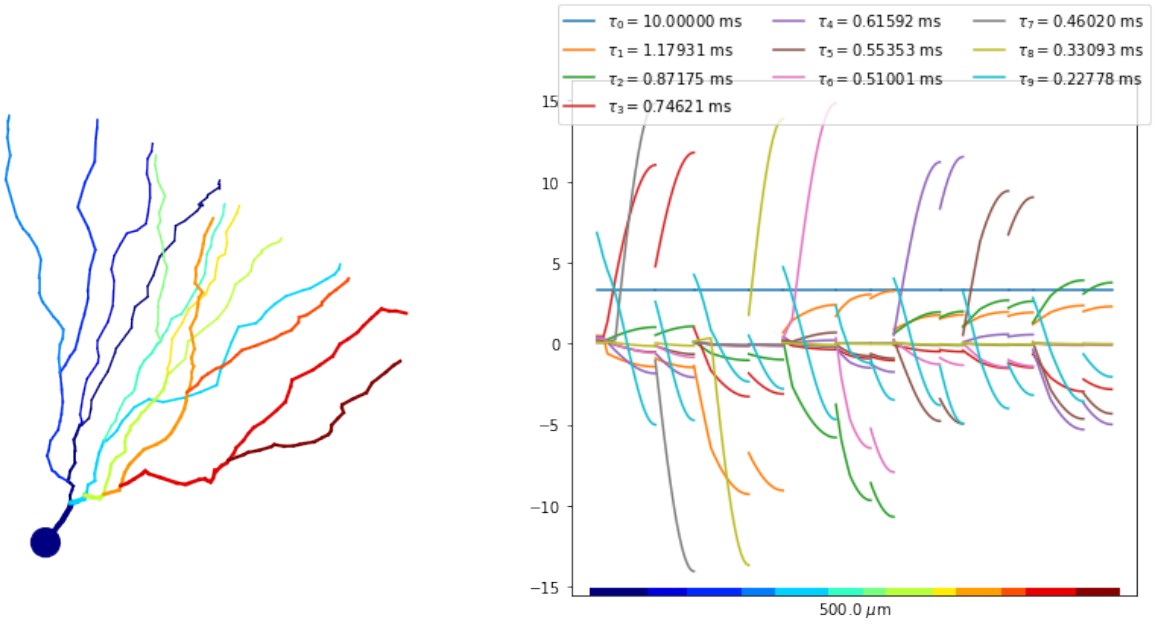
The spatial functions are returned as the second argument of `SOVTree.getSOVMatrics()` or `SOVTree.getImportantModes()`. The first dimension signifies the index of the SOV term and the second dimension the index of the location.

```
[7]: print(``phimat` is evaluated for %d SOV terms and at %d locations, stored under \
     ↪'xaxis\''%phimat_.shape)
```

```
`phimat` is evaluated for 37 SOV terms and at 223 locations, stored under 'xaxis'
```

Finally, we may plot these spatial functions by unraveling the morphology on a one-dimensional axis.

[8]:
```
pl.figure(figsize=(12,6))
colours = list(pl.rcParams['axes.prop_cycle'].by_key()['color'])
# plot the morphology
ax0 = pl.subplot(121)
sov_tree.setNodeColors()
sov_tree.plot2DMorphology(ax0, cs='x_color', cmap=pl.get_cmap('jet'), plotargs={'lw':
↪3})
# plot the spatial mode functions
ax1 = pl.subplot(122)
for ii, phi in enumerate(phimat_[:10]):
    sov_tree.plot1D(ax1, phi.real, c=colours[ii%len(colours)],
                    label=r'$\tau_'+'{%d} = %.5f$ ms'%(ii, 1./alphas_[ii].real))
sov_tree.colorXAxis(ax1, cmap=pl.get_cmap('jet'))
ax1.legend(loc='lower center', ncol=3, bbox_to_anchor=(.5,.9))

pl.tight_layout()
pl.show()
```



6.3 Reference

Release 0.9.1

Date Mar 25, 2021

6.3.1 Abstract Trees

Basic tree

class STree(*root=None*)

A simple tree for use with a simple Node (*neat.SNode*).

Generic implementation of a tree structure as a linked list extended with some convenience functions

Parameters **root** (*neat.SNode*, optional) – The root of the tree, default is `None` which creates an empty tree

Variables **root** (*neat.SNode*) – The root of the tree

STree.__getitem__(index, **kwargs)	Returns the node with given index, if no such node is in the tree, None is returned.
STree.__len__([node])	Return the number of nodes in the tree.
STree.__iter__([node])	Iterate over the nodes in the subtree of the given node.
STree.__str__([node])	Generate a string of the subtree of the given node.
STree.__copy__([new_tree])	Fill the `new_tree` with it's corresponding nodes in the same structure as `self`, and copies all node variables that both tree classes have in common
STree.checkOrdered()	Check if the indices of the tree are number in the same order as they appear in the iterator
STree.getNodes([recompute_flag])	Build a list of all the nodes in the tree
STree.nodes	Build a list of all the nodes in the tree
STree.gatherNodes(node)	Build a list of all the nodes in the subtree of the provided node
STree.getLeafs([recompute_flag])	Get all leaf nodes in the tree.
STree.leafs	Get all leaf nodes in the tree.
STree.isLeaf(node)	Check if input node is a leaf of the tree
STree.root	
STree.isRoot(node)	Check if input node is root of the tree.
STree.addNodeWithParentFromIndex(node_index, ...)	Create a node with the given index and add it to the tree under a specific parent node.
STree.addNodeWithParent(node, pnode)	Add a node to the tree under a specific parent node
STree.softRemoveNode(node)	Remove a node and its subtree from the tree by deleting the reference to it in its parent.
STree.removeNode(node)	Remove a node as well as its subtree from the tree
STree.removeSingleNode(node)	Remove a single node from the tree.
STree.insertNode(node, pnode[, pcnodes])	Insert a node in the tree as a child of a specified parent.
STree.resetIndices([n])	Resets the indices in the order they appear in a depth-first iteration
STree.getSubTree(node[, new_tree])	Get the subtree of the specified node.
STree.depthOfNode(node)	compute the depth of the node (number of edges between node and root)
STree.degreeOfNode(node)	Compute the degree (number of leafs in its subtree) of a node.
STree.orderOfNode(node)	Compute the order (number of bifurcations from the root) of a node.
STree.pathToRoot(node)	Return the path from a given node to the root
STree.pathBetweenNodes(from_node, to_node)	Inclusive path from `from_node` to `to_node`.

continues on next page

Table 1 – continued from previous page

STree.pathBetweenNodesDepthFirst(from_node, ...)	Inclusive path from `from_node` to `to_node`, ginven in a depth- first ordering.
STree.getNodesInSubtree(ref_node[, subtree_root])	Returns the nodes in the subtree that contains the given reference nodes and has the given subtree root as root.
STree.sisterLeafs(node)	Find the leafs that are in the subtree of the nearest bifurcation node up from the input node.
STree.upBifurcationNode(node[, cnode])	Find the nearest bifurcation node up (towards root) from the input node.
STree.downBifurcationNode(node)	Find the nearest bifurcation node down (towards leafs) from the input node.
STree.getBifurcationNodes(nodes)	Get the bifurcation nodes in bewteen the provided input nodes
STree.getNearestNeighbours(node, nodes)	Find the nearest neighbours of `node` in `nodes`.
STree.__copy__([new_tree])	Fill the `new_tree` with it's corresponding nodes in the same structure as `self`, and copies all node variables that both tree classes have in common

neat.STree.__getitem__

STree.**__getitem__**(*index*, ***kwargs*)

Returns the node with given index, if no such node is in the tree, None is returned.

Parameters **index** (*int*) – the index of the node to be found

Returns

Return type *neat.SNode* or None

neat.STree.__len__

STree.**__len__**(*node=None*)

Return the number of nodes in the tree. If an input node is specified, the number of nodes in the subtree of the input node is returned

Parameters **node** (*neat.SNode* (optional)) – The starting node. Defaults to root

Returns

Return type int

neat.STree.__iter__

STree.**__iter__**(*node=None*, ***kwargs*)

Iterate over the nodes in the subtree of the given node.

Beware, if the given node is not in the tree, it will simply iterate over the subtree of the given node.

Parameters **node** (*neat.SNode* (optional)) – The starting node. Defaults to the root

neat.STree.__str__

STree.**__str__**(*node=None*)
Generate a string of the subtree of the given node.

Beware, if the given node is not in the tree, it will simply iterate over the subtree of the given node.

Parameters **node** (*neat.SNode* (optional)) – The starting node. Defaults to the root

neat.STree.__copy__

STree.**__copy__**(*new_tree=None*)
Fill the `new_tree` with it's corresponding nodes in the same structure as `self`, and copies all node variables that both tree classes have in common

Parameters **new_tree** (*neat.STree* or derived class (default is `None`)) – the tree class in which the `self` is copied. If `None`, returns a copy of `self`.

Returns

Return type The new tree instance

neat.STree.checkOrdered

STree.**checkOrdered**()
Check if the indices of the tree are number in the same order as they appear in the iterator

neat.STree.getNodes

STree.**getNodes**(*recompute_flag=1*)
Build a list of all the nodes in the tree

Parameters **recompute_flag** (*bool*) – whether or not to re-evaluate the node list

Returns

Return type list of `Snode`

neat.STree.nodes

property STree.**nodes**
Build a list of all the nodes in the tree

Parameters **recompute_flag** (*bool*) – whether or not to re-evaluate the node list

Returns

Return type list of `Snode`

neat.STree.gatherNodes

STree.**gatherNodes**(*node*)
Build a list of all the nodes in the subtree of the provided node

Parameters **node** (Snode) – starting point node

Returns

Return type list of Snode

neat.STree.getLeafs

STree.**getLeafs**(*recompute_flag=1*)
Get all leaf nodes in the tree.

Parameters **recompute_flag** (*bool*) – Whether to force recomputing the leaf list. Defaults to 1.

neat.STree.leafs

property STree.**leafs**
Get all leaf nodes in the tree.

Parameters **recompute_flag** (*bool*) – Whether to force recomputing the leaf list. Defaults to 1.

neat.STree.isLeaf

STree.**isLeaf**(*node*)
Check if input node is a leaf of the tree

Parameters **node** (*neat.SNode*) –

neat.STree.root

property STree.**root**

neat.STree.isRoot

STree.**isRoot**(*node*)
Check if input node is root of the tree.

Parameters **node** (*neat.SNode*) –

neat.STree.addNodeWithParentFromIndex

STree.**addNodeWithParentFromIndex**(*node_index*, *pnode*, **args*, ***kwargs*)

Create a node with the given index and add it to the tree under a specific parent node.

Parameters

- **node_index** (*int*) – index of the new node
- **pnode** (*neat.SNode*) – parent node of the newly added node

Raises **ValueError** – if node_index is already in the tree

neat.STree.addNodeWithParent

STree.**addNodeWithParent**(*node*, *pnode*)

Add a node to the tree under a specific parent node

Parameters

- **node** (*neat.SNode*) – node to be added
- **pnode** (*neat.SNode*) – parent node of the newly added node

neat.STree.softRemoveNode

STree.**softRemoveNode**(*node*)

Remove a node and its subtree from the tree by deleting the reference to it in its parent. Internally, the node and its linked subtree are not changed

Parameters **node** (*neat.SNode*) – node to be removed

neat.STree.removeNode

STree.**removeNode**(*node*)

Remove a node as well as its subtree from the tree

Parameters **node** (*neat.SNode*) – node to be removed

neat.STree.removeSingleNode

STree.**removeSingleNode**(*node*)

Remove a single node from the tree. The nodes' children become the children of the nodes' parent.

Parameters **node** (*neat.SNode*) – node to be removed

neat.STree.insertNode

STree.**insertNode**(*node*, *pnode*, *pcnodes=[]*)

Insert a node in the tree as a child of a specified parent. The original children of the parent that will become children of the node are specified in the `pcnodes` list

Parameters

- **node** (*neat.SNode*) – the node that is to be inserted
- **pnode** (*neat.SNode*) – the node that will become parent of the node that is to be inserted
- **pcnodes** (list of *neat.SNode*) – the current children of the pnode that will become children of the node

neat.STree.resetIndices

STree.**resetIndices**(*n=0*)

Resets the indices in the order they appear in a depth-first iteration

neat.STree.getSubTree

STree.**getSubTree**(*node*, *new_tree=None*)

Get the subtree of the specified node. The root of the subtree is a new node with the same children as the original node, but None instead of a parent.

Parameters **node** (*neat.SNode*) – root of the sub tree

Returns Subtree of with `node` as root

Return type *neat.STree*

neat.STree.depthOfNode

STree.**depthOfNode**(*node*)

compute the depth of the node (number of edges between node and root)

Parameters **node** (*neat.SNode*) –

Returns depth of the node

Return type int

neat.STree.degreeOfNode

STree.**degreeOfNode**(*node*)

Compute the degree (number of leafs in its subtree) of a node.

Parameters **node** (*neat.SNode*) –

neat.STree.orderOfNode

STree.**orderOfNode**(*node*)
Compute the order (number of bifurcations from the root) of a node.

Parameters **node** (*neat.SNode*) –

neat.STree.pathToRoot

STree.**pathToRoot**(*node*)
Return the path from a given node to the root

Parameters: node: *neat.SNode*

Returns List of nodes from `node` to root. First node is the input node and last node is the root

Return type list of *neat.SNode*

neat.STree.pathBetweenNodes

STree.**pathBetweenNodes**(*from_node*, *to_node*)
Inclusive path from `from_node` to `to_node`.

Parameters

- **from_node** (*neat.SNode*) –
- **to_node** (*neat.SNode*) –

Returns List of nodes representing the direct path between `from_node` and `to_node`, which are respectively the first and last nodes in the list.

Return type list of *neat.SNode*

neat.STree.pathBetweenNodesDepthFirst

STree.**pathBetweenNodesDepthFirst**(*from_node*, *to_node*)
Inclusive path from `from_node` to `to_node`, ginven in a depth- first ordering.

Parameters

- **from_node** (*neat.SNode*) –
- **to_node** (*neat.SNode*) –

Returns List of nodes representing the direct path between `from_node` and `to_node`, which are respectively the first and last nodes in the list.

Return type list of *neat.SNode*

neat.STree.getNodesInSubtree

STree.**getNodesInSubtree**(*ref_node*, *subtree_root=None*)

Returns the nodes in the subtree that contains the given reference nodes and has the given subtree root as root. If the subtree root is not provided, the subtree of the first child node of the root on the path to the reference node is given (plus the root)

Parameters

- **ref_node** (*neat.SNode*) – the reference node that is in the subtree
- **subtree_root** (*neat.SNode*) – what is to be the root of the subtree. If this node is not on the path from reference node to root, a ValueError is raised

Returns List of all nodes in the subtree. It's root is in the first position

Return type list of *neat.SNode*

neat.STree.sisterLeafs

STree.**sisterLeafs**(*node*)

Find the leafs that are in the subtree of the nearest bifurcation node up from the input node.

Parameters **node** (*neat.SNode*) – Starting node for search

Returns

- **node** (*neat.SNode*) – the bifurcation node
- **sister_leafs** (list of *neat.SNode*) – The first element is the input node. The others are the leafs of the subtree emanating from the bifurcation node that are not in the subtree from the input node.
- **corresponding_children** (list of *neat.SNode*) – The children of the bifurcation node. If the number of leafs `sister_leafs` is the same as the number of `corresponding_children`, the subtree of each element of `corresponding_children` has exactly one leaf, the corresponding element in `sister_leafs`

neat.STree.upBifurcationNode

STree.**upBifurcationNode**(*node*, *cnode=None*)

Find the nearest bifurcation node up (towards root) from the input node.

Parameters

- **node** (*neat.SNode*) – Starting node for search
- **cnode** (*neat.SNode*) – For recursion, don't change default

Returns

- **node** (*neat.SNode*) – the bifurcation node
- **cnode** (*neat.SNode*) – The bifurcation node's child on the path to the input node.

neat.STree.downBifurcationNode

STree.**downBifurcationNode**(*node*)

Find the nearest bifurcation node down (towards leafs) from the input node.

Parameters **node** (*neat.SNode*) – Starting node for search

Returns **node** – the bifurcation node

Return type *neat.SNode*

neat.STree.getBifurcationNodes

STree.**getBifurcationNodes**(*nodes*)

Get the bifurcation nodes in bewteen the provided input nodes

Parameters **nodes** (list of *neat.SNode*) – the input nodes

Returns the bifurcation nodes

Return type list of *neat.SNode*

neat.STree.getNearestNeighbours

STree.**getNearestNeighbours**(*node*, *nodes*)

Find the nearest neighbours of `node` in `nodes`. If `nodes` contains `node`, it is excluded from the search.

When a node in the up-direction is a bifurcation node and in `nodes`, nodes in its other subtree are excluded from the search

!!! Untested

Parameters

- **node** (*neat.SNode*) – node for which the nearest neighbours are sought
- **nodes** (list of *neat.SNode*) – list in which nearest neighbours of `node` are sought

class SNode(*index*)

Simple Node for use with a simple Tree (*neat.STree*)

Parameters **index** (*int*) – index of the node

Variables

- **index** (*int*) – index of the node
- **parent_node** (*neat.SNode* or `None`) – parent of node, `None` means node is root
- **child_nodes** (list of *neat.SNode*) – child nodes of `self`, empty list means node is leaf
- **content** (*dict*) – arbitrary items can be stored at the node

Compartment Tree

class CompartmentTree(*root=None*)

Abstract tree that implements physiological parameters for reduced compartmental models. Also implements the matrix algebra to fit physiological parameters to impedance matrices

CompartmentTree.addCurrent(channel, e_rev)	Add an ion channel current to the tree
CompartmentTree.setExpansionPoints(...)	Set the choice for the state variables of the ion channel around which to linearize.
CompartmentTree.setEEq(e_eq[, indexing])	Set the equilibrium potential at all nodes on the compartment tree
CompartmentTree.getEEq([indexing])	Get the equilibrium potentials at each node.
CompartmentTree.fitEL()	Fit the leak reversal potential to obtain the stored equilibirum potentials as resting membrane potential
CompartmentTree.getEquivalentLocs()	Get list of fake locations in the same order as original list of locations to which the compartment tree was fitted.
CompartmentTree.calcImpedanceMatrix([freqs, ...])	Constructs the impedance matrix of the model for each frequency provided in *freqs*.
CompartmentTree.calcConductanceMatrix([indexing])	Constructs the conductance matrix of the model
CompartmentTree.calcSystemMatrix([freqs, ...])	Constructs the matrix of conductance and capacitance terms of the model for each frequency provided in `freqs`.
CompartmentTree.calcEigenvalues([indexing])	Calculates the eigenvalues and eigenvectors of the passive system
CompartmentTree.computeGMC(z_mat_arg[, ...])	Fit the models' membrane and coupling conductances to a given steady state impedance matrix.
CompartmentTree.computeGChanFromImpedance(...)	Fit the conductances of multiple channels from the given impedance matrices, or store the feature matrix and target vector for later use (see *action*).
CompartmentTree.computeGSingleChanFromImpedance(...)	Fit the conductances of a single channel from the given impedance matrices, or store the feature matrix and target vector for later use (see *action*).
CompartmentTree.computeC(alphas, phimat[, ...])	Fit the capacitances to the eigenmode expansion
CompartmentTree.resetFitData()	Delete all stored feature matrices and and target vectors.
CompartmentTree.runFit()	Run a linear least squares fit for the conductances concentration mechanisms.
CompartmentTree.computeFakeGeometry([...])	Computes a fake geometry so that the neuron model is a reduced compurtmental model
CompartmentTree.plotDendrogram(ax[, ...])	Generate a dendrogram of the NET

neat.CompartmentTree.addCurrent

CompartmentTree.**addCurrent**(*channel*, *e_rev*)

Add an ion channel current to the tree

Parameters

- **channel_name** (*string*) – The name of the channel type
- **e_rev** (*float*) – The reversal potential of the ion channel [mV]

neat.CompartmentTree.setExpansionPoints

CompartmentTree.**setExpansionPoints**(*expansion_points*)

Set the choice for the state variables of the ion channel around which to linearize.

Note that when adding an ion channel to the tree, the default expansion point setting is to linearize around the asymptotic values for the state variables at the equilibrium potential store in *self.e_eq*. Hence, this function only needs to be called to change that setting.

Parameters **expansion_points** (dict {*channel_name*: None or dict}) – dictionary with as keys *channel_name* the name of the ion channel and as value its expansion point

neat.CompartmentTree.setEEq

CompartmentTree.**setEEq**(*e_eq*, *indexing='locs'*)

Set the equilibrium potential at all nodes on the compartment tree

Parameters

- **e_eq** (*float or np.array of floats*) – The equilibrium potential(s). If a float, the same potential is set at every node. If a numpy array, must have the same length as *self*
- **indexing** (*'locs' or 'tree'*) – The ordering of the equilibrium potentials. If 'locs', assumes the equilibrium potentials are in the order of the list of locations to which the tree is fitted. If 'tree', assumes they are in the order of which nodes appear during iteration

neat.CompartmentTree.getEEq

CompartmentTree.**getEEq**(*indexing='locs'*)

Get the equilibrium potentials at each node.

Parameters **indexing** (*'locs' or 'tree'*) – The ordering of the returned array. If 'locs', returns the array in the order of the list of locations to which the tree is fitted. If 'tree', returns the array in the order in which nodes appear during iteration

neat.CompartmentTree.fitEL

CompartmentTree.**fitEL**()
Fit the leak reversal potential to obtain the stored equilibirum potentials as resting membrane potential

neat.CompartmentTree.getEquivalentLocs

CompartmentTree.**getEquivalentLocs**()
Get list of fake locations in the same order as original list of locations to which the compartment tree was fitted.

Returns Tuple has the form *(node.index, .5)*

Return type list of tuple

neat.CompartmentTree.calcImpedanceMatrix

CompartmentTree.**calcImpedanceMatrix**(*freqs=0.0*, *channel_names=None*, *indexing='locs'*, *use_conc=False*)
Constructs the impedance matrix of the model for each frequency provided in *freqs*. This matrix is evaluated at the equilibrium potentials stored in each node

Parameters

- **freqs** (*np.array (dtype = complex) or float*) – Frequencies at which the matrix is evaluated [Hz]
- **channel_names** (None (default) or *list* of *str*) – The channels to be included in the matrix. If None, all channels present on the tree are included in the calculation
- **use_conc** (*bool*) – wheter or not to use the concentration dynamics
- **indexing** (*'tree' or 'locs'*) – Whether the indexing order of the matrix corresponds to the tree nodes (order in which they occur in the iteration) or to the locations on which the reduced model is based

Returns The first dimension corresponds to the frequency, the second and third dimension contain the impedance matrix for that frequency

Return type *np.ndarray* (ndim = 3, dtype = complex)

neat.CompartmentTree.calcConductanceMatrix

CompartmentTree.**calcConductanceMatrix**(*indexing='locs'*)
Constructs the conductance matrix of the model

Returns the conductance matrix

Return type *np.ndarray* (dtype = float, ndim = 2)

neat.CompartmentTree.calcSystemMatrix

CompartmentTree.**calcSystemMatrix**(*freqs=0.0*, *channel_names=None*, *with_ca=True*, *use_conc=False*, *indexing='locs'*)

Constructs the matrix of conductance and capacitance terms of the model for each frequency provided in `freqs`. this matrix is evaluated at the equilibrium potentials stored in each node

Parameters

- **freqs** (np.array (dtype = complex) or float (default `0.`)) – Frequencies at which the matrix is evaluated [Hz]
- **channel_names** (*None* (default) or *list* of *str*) – The channels to be included in the matrix. If *None*, all channels present on the tree are included in the calculation
- **with_ca** (*bool*) – Whether or not to include the capacitive currents
- **use_conc** (*bool*) – wheter or not to use the concentration dynamics
- **indexing** (*`'tree' or 'locs'`*) – Whether the indexing order of the matrix corresponds to the tree nodes (order in which they occur in the iteration) or to the locations on which the reduced model is based

Returns The first dimension corresponds to the frequency, the second and third dimension contain the impedance matrix for that frequency

Return type *np.ndarray* (`ndim = 3, dtype = complex`)

neat.CompartmentTree.calcEigenvalues

CompartmentTree.**calcEigenvalues**(*indexing='tree'*)

Calculates the eigenvalues and eigenvectors of the passive system

Returns

- *np.ndarray (ndim = 1, dtype = complex)* – the eigenvalues
- *np.ndarray (ndim = 2, dtype = complex)* – the right eigenvector matrix
- **indexing** (*'tree' or 'locs'*) – Whether the indexing order of the matrix corresponds to the tree nodes (order in which they occur in the iteration) or to the locations on which the reduced model is based

neat.CompartmentTree.computeGMC

CompartmentTree.**computeGMC**(*z_mat_arg*, *e_eqs=None*, *channel_names=['L']*)

Fit the models' membrane and coupling conductances to a given steady state impedance matrix.

Parameters

- **z_mat_arg** (*`np.ndarray (ndim = 2, dtype = float or complex) or`*) – list of np.ndarray (ndim = 2, dtype = float or complex) If a single array, represents the steady state impedance matrix, If a list of arrays, represents the steady state impedance matrices for each equilibrium potential in `e_eqs`
- **e_eqs** (*`np.ndarray (ndim = 1, dtype = float) or float`*) – The equilibrium potentials in each compartment for each evaluation of `z_mat`

- **channel_names** (*list of string (defaults to ['L'])*) – Names of the ion channels that have been included in the impedance matrix calculation and for whom the conductances are fit. Default is only leak conductance

neat.CompartmentTree.computeGChanFromImpedance

CompartmentTree.**computeGChanFromImpedance**(*channel_names*, *z_mat*, *e_eq*, *freqs*, *sv=None*, *weight=1.0*, *all_channel_names=None*, *other_channel_names=None*, *action='store'*)

Fit the conductances of multiple channels from the given impedance matrices, or store the feature matrix and target vector for later use (see *action*).

Parameters

- **channel_names** (*list of str*) – The names of the ion channels whose conductances are to be fitted
- **z_mat** (*np.ndarray (ndim=3)*) – The impedance matrix to which the ion channel is fitted. Shape is `(F, N, N)` with `N` the number of compartments and `F` the number of frequencies at which the matrix is evaluated
- **e_eq** (*float*) – The equilibirum potential at which the impedance matrix was computed
- **freqs** (*np.array*) – The frequencies at which *z_mat* is computed (shape is `(F,)`)
- **sv** (*dict {channel_name: np.ndarray} (optional)*) – The state variable expansion point. If `np.ndarray`, assumes it is the expansion point of the channel that is fitted. If dict, the expansion points of multiple channels can be specified. An empty dict implies the asymptotic points derived from the equilibrium potential
- **weight** (*float*) – The relative weight of the feature matrices in this part of the fit
- **all_channel_names** (list of str or `None`) – The names of all channels whose conductances will be fitted in a single linear least squares fit
- **other_channel_names** (list of str or `None` (default)) – List of channels present in *z_mat*, but whose conductances are already fitted. If `None` and 'L' is not in *all_channel_names*, sets *other_channel_names* to 'L'
- **action** (*'fit', 'store' or 'return'*) – If 'fit', fits the conductances for this feature matrix and target vector for directly; only based on *z_mat*; nothing is stored. If 'store', stores the feature matrix and target vector to fit later on. Relative weight in fit will be determined by *weight*. If 'return', returns the feature matrix and target vector. Nothing is stored

neat.CompartmentTree.computeGSingleChanFromImpedance

CompartmentTree.**computeGSingleChanFromImpedance**(*channel_name*, *z_mat*, *e_eq*, *freqs*, *sv=None*, *weight=1.0*, *all_channel_names=None*, *other_channel_names=None*, *action='store'*)

Fit the conductances of a single channel from the given impedance matrices, or store the feature matrix and target vector for later use (see *action*).

Parameters

- **channel_name** (*str*) – The name of the ion channel whose conductances are to be fitted

- **z_mat** (*np.ndarray (ndim=3)*) – The impedance matrix to which the ion channel is fitted. Shape is `(F, N, N)` with `N` the number of compartments and `F` the number of frequencies at which the matrix is evaluated
- **e_eq** (*float*) – The equilibirum potential at which the impedance matrix was computed
- **freqs** (*np.array*) – The frequencies at which *z_mat* is computed (shape is `(F,)`)
- **sv** (*dict or nested dict of float or np.array, or None (default)*) – The state variable expansion point. If simple dict, assumes it is the expansion point of the channel that is fitted. If nested dict, the expansion points of multiple channels can be specified. `None` implies the asymptotic point derived from the equilibrium potential
- **weight** (*float*) – The relative weight of the feature matrices in this part of the fit
- **all_channel_names** (list of str or `None`) – The names of all channels whose conductances will be fitted in a single linear least squares fit
- **other_channel_names** (list of str or `None` (default)) – List of channels present in *z_mat*, but whose conductances are already fitted. If `None` and 'L' is not in *all_channel_names*, sets *other_channel_names* to 'L'
- **action** (*'fit', 'store' or 'return'*) – If 'fit', fits the conductances for this feature matrix and target vector for directly; only based on *z_mat*; nothing is stored. If 'store', stores the feature matrix and target vector to fit later on. Relative weight in fit will be determined by *weight*. If 'return', returns the feature matrix and target vector. Nothing is stored

neat.CompartmentTree.computeC

`CompartmentTree.`**computeC**(*alphas*, *phimat*, *weights=None*, *tau_eps=5.0*)

Fit the capacitances to the eigenmode expansion

Parameters

- **alphas** (*np.ndarray of float or complex (shape=(K,))*) – The eigenmode inverse timescales (1/s)
- **phimat** (*np.ndarray of float or complex (shape=(K,C))*) – The eigenmode vectors (C the number of compartments)
- **weights** (*np.ndarray (shape=(K,)) or None*) – The weights given to each eigenmode in the fit

neat.CompartmentTree.resetFitData

`CompartmentTree.`**resetFitData**()

Delete all stored feature matrices and and target vectors.

neat.CompartmentTree.runFit

CompartmentTree.**runFit**()

Run a linear least squares fit for the conductances concentration mechanisms. The obtained conductances are stored on each node. All stored feature matrices and and target vectors are deleted.

neat.CompartmentTree.computeFakeGeometry

CompartmentTree.**computeFakeGeometry**(*fake_c_m=1.0*, *fake_r_a=9.999999999999999e-05*, *factor_r_a=1e-06*, *delta=1e-14*, *method=2*)

Computes a fake geometry so that the neuron model is a reduced compurtmental model

Parameters

- **fake_c_m** (*float [uF / cm^2]*) – fake membrane capacitance value used to compute the surfaces of the compartments
- **fake_r_a** (*float [MOhm * cm]*) – fake axial resistivity value, used to evaluate the lengths of each section to yield the correct coupling constants

Returns **radii, lengths** – The radii, lengths, resp. surfaces for the section in NEURON. Array index corresponds to NEURON index

Return type np.array of floats [cm]

Raises **AssertionError** – If the node indices are not ordered consecutively when iterating

neat.CompartmentTree.plotDendrogram

CompartmentTree.**plotDendrogram**(*ax*, *plotargs={}*, *labelargs={}*, *textargs={}*, *nodelabels={}*, *bbox=None*, *y_max=None*)

Generate a dendrogram of the NET

Parameters

- **ax** (*matplotlib.axes*) – the axes object in which the plot will be made
- **plotargs** (*dict (string : value)*) – keyword args for the matplotlib plot function, specifies the line properties of the dendrogram
- **labelargs** (*dict (string : value)*) – keyword args for the matplotlib plot function, specifies the marker properties for the node points. Or dict with keys node indices, and with values dicts with keyword args for the matplotlib function that specify the marker properties for specific node points. The entry under key -1 specifies the properties for all nodes not explicitly in the keys.
- **textargs** (*dict (string : value)*) – keyword args for matplotlib textproperties
- **nodelabels** (*dict (int: string) or None*) – labels of the nodes. If None, nodes are named by default according to their location indices. If empty dict, no labels are added.
- **y_max** (*int, float or None*) – specifies the y-scale. If None, the scale is computed from `self`. By default, y=1 is added for each child of a node, so if y_max is smaller than the depth of the tree, part of it will not be plotted

class CompartmentNode(*index*, *loc_ind=None*, *ca=1.0*, *g_c=0.0*, *g_l=0.01*, *e_eq=- 75.0*)

Implements a node for *CompartmentTree*

Variables

- **ca** (*float*) – capacitance of the compartment (uF)
- **g_l** (*float*) – leak conductance at the compartment (uS)
- **g_c** (*float*) – Coupling conductance of compartment with parent compartment (uS). Ignore if node is the root
- **e_eq** (*float*) – equilibrium potential at the compartment
- **currents** (*dict {str: [g_bar, e_rev]}*) – dictionary with as keys the channel names and as elements lists of length two with contain the maximal conductance (uS) and the channels' reversal potential in (mV)
- **concmechs** (dict {str: *neat.channels.concmechs.ConcMech*}) – dictionary with as keys the ion names and as values the concentration mechanisms governing the concentration of each ion channel
- **expansion_points** (*dict {str: np.ndarray}*) – dictionary with as keys the channel names and as elements the state variables of the ion channel around which to compute the linearizations

Neural Evaluation Tree

class NET(*root=None*)

Abstract tree class that implements the Neural Evaluation Tree (Wybo et al., 2019), representing the spatial voltage as a number of voltage components present at different spatial scales.

NET.getLocInds([sroot])	Get the indices of the locations a subtree integrates
NET.getLeafLocNode(loc_ind)	Get the node for which `loc_ind` is a new location
NET.setNewLocInds()	Set the new location indices in a tree
NET.getReducedTree(loc_inds[, indexing])	Construct a reduced tree where only the locations index by ``loc_inds'' are retained
NET.calcTotalImpedance(node)	Compute the total impedance associated with a node.
NET.calcIZ(loc_inds)	compute I_Z between any pair of locations in `loc_inds`
NET.calcIZMatrix()	compute the Iz matrix for all locations present in the tree
NET.calcImpedanceMatrix()	Compute the impedance matrix approximation associated with the NET
NET.calcImpMat()	Compute the impedance matrix approximation associated with the NET
NET.getCompartmentalization(Iz[, returntype])	Returns a compartmentalization for the NET tree where each pair of compartments is separated by an Iz of at least `Iz`.
NET.plotDendrogram(ax[, plotargs, ...])	Generate a dendrogram of the NET

neat.NET.getLocInds

NET.**getLocInds**(*sroot=None*)
Get the indices of the locations a subtree integrates

Parameters **sroot** (*neat.NETNode*, int or None) – Root of the subtree, or index of the root. If None, subtree is the whole tree.

Returns **loc_inds**

Return type indices of locations

neat.NET.getLeafLocNode

NET.**getLeafLocNode**(*loc_ind*)
Get the node for which loc_ind is a new location

Parameters **loc_ind** (*int*) – index of the location

Returns

Return type *NETNode*

neat.NET.setNewLocInds

NET.**setNewLocInds**()
Set the new location indices in a tree

neat.NET.getReducedTree

NET.**getReducedTree**(*loc_inds*, *indexing='NET eval'*)
Construct a reduced tree where only the locations index by ``loc_inds" are retained

Parameters

- **loc_inds** (*iterable of ints*) – the indices of the locations that are to be retained
- **indexing** (*'NET eval' or 'locs'*) – if 'NET eval', indexing of NETNode.loc_inds will be taken to be the indices of locations for which the full NET is evaluated. Otherwise will be indices of the input loc_inds

neat.NET.calcTotalImpedance

NET.**calcTotalImpedance**(*node*)
Compute the total impedance associated with a node. I.e. the sum of all impedances on the path from node to root

Parameters **node** (*SNode*) –

Returns total impedance

Return type float

neat.NET.calcIZ

NET.**calcIZ**(*loc_inds*)
compute I_Z between any pair of locations in `loc_inds`

Parameters **loc_inds** (*iterable of ints*) – the indices of locations between which I_Z has to be evaluated

Returns **float or dict of tuple** – Returns a float if the number of location indices is two, otherwise a dictionary with location pairs (smallest is listed first) as keys and I_Z values as values

Return type float

neat.NET.calcIZMatrix

NET.**calcIZMatrix**()
compute the Iz matrix for all locations present in the tree

Returns The Iz matrix

Return type np.ndarray of float

neat.NET.calcImpedanceMatrix

NET.**calcImpedanceMatrix**()
Compute the impedance matrix approximation associated with the NET

Returns the impedance matrix approximation

Return type np.ndarray (ndim = 2)

neat.NET.calcImpMat

NET.**calcImpMat**()
Compute the impedance matrix approximation associated with the NET

Returns the impedance matrix approximation

Return type np.ndarray (ndim = 2)

neat.NET.getCompartmentalization

NET.**getCompartmentalization**(*Iz*, *returntype='node index'*)
Returns a compartmentalization for the NET tree where each pair of compartments is separated by an Iz of at least `Iz`. The compartmentalization is coded as a list of list, each sublist representing a the nodes closest to the root associated with the compartment.

Parameters

- **Iz** (*float*) – the minimum Iz separating the compartments
- **returntype** (*str ('node index', 'node')*) – either returns the node indices or the node objects

Returns the compartments

Return type list of lists

neat.NET.plotDendrogram

NET.**plotDendrogram**(*ax*, *plotargs={}*, *labelargs={}*, *textargs={}*, *incolors={}*, *inlabels={}*, *nodelabels={}*, *cs_comp={}*, *cmap=None*, *z_max=None*, *add_scalebar=True*)

Generate a dendrogram of the NET

Parameters

- **ax** (*matplotlib.axes*) – the axes object in which the plot will be made
- **plotargs** (*dict (string : value)*) – keyword args for the matplotlib plot function, specifies the line properties of the dendrogram
- **labelargs** (*dict (string : value)*) – keyword args for the matplotlib plot function, specifies the marker properties for the node points. Or dict with keys node indices, and with values dicts with keyword args for the matplotlib function that specify the marker properties for specific node points. The entry under key -1 specifies the properties for all nodes not explicitly in the keys.
- **textargs** (*dict (string : value)*) – keyword args for matplotlib textproperties
- **incolors** (*dict (int : string)*) – dict with locinds as keys and colors as values
- **inlabels** (*dict (int : string)*) – dict with locinds as keys and label strings as values
- **nodelabels** (*dict (int: string) or None*) – labels of the nodes. If None, nodes are named by default according to their location indices. If empty dict, no labels are added.
- **cs_comp** (*dict (int : float)*) – dict with node inds as keys and compartment colors as values
- **z_max** (*float or None*) – specifies the y-scale. If None, the scale is computed from *self*
- **add_scalebar** (*bool*) – whether or not to add a scale bar

class NETNode(*index*, *loc_inds*, *newloc_inds=[]*, *z_kernel=None*)

Node associated with *neat.NET*.

Variables

- **loc_inds** (*list of int*) – The inidices of locations which the node integrates
- **newloc_inds** (*list of int*) – The locations for which the node is the most local component to integrate them
- **z_kernel** (*neat.Kernel*) – The impedance kernel with which the node integrates inputs
- **z_bar** (*float*) – The steady state impedance associated with the impedance kernel

class Kernel(*kernel*)

Implements a kernel as a superposition of exponentials:

$$k(t) = \sum_n c_n e^{-a_n t}$$

Kernels can be added and subtracted, as this class overloads the __add__ and __subtract__ functions.

They can be evaluated as a function of time by calling the object with a time array.

They can be evaluated in the Fourrier domain with *Kernel.ft*

Parameters **kernel** (*dict, float, neat.Kernel, tuple or list*) – If dict, has the form {'a': *np.array*, 'c': *np.array*}. If float, sets *c* single exponential prefactor and assumes *a* is 1 kHz. If *neat.Kernel*, copies the object. If tuple or list, sets 'a' as first element and 'c' as last element.

Variables

- **a** (*np.array of float or complex*) – The exponential coefficients (kHz)
- **c** (*np.array of float or complex*) – The exponential prefactors

Kernel.k_bar	The total surface under the kernel
Kernel.t(t_arr)	Evaluates the kernel in the time domain
Kernel.ft(s_arr)	Evaluates the kernel in the Fourrier domain

neat.Kernel.k_bar

property Kernel.**k_bar**
The total surface under the kernel

neat.Kernel.t

Kernel.**t**(*t_arr*)
Evaluates the kernel in the time domain

Parameters **t_arr** (*np.array of float*) – the time array at which the kernel is evaluated

Returns the temporal kernel

Return type np.array of float

neat.Kernel.ft

Kernel.**ft**(*s_arr*)
Evaluates the kernel in the Fourrier domain

Parameters **s_arr** (*np.array of complex*) – The frequencies (Hz) at which the kernel is to be evaluated

Returns The Fourrier transform of the kernel

Return type np.array of complex

Simulate reduced compartmental models

class NeuronCompartmentTree(*t_calibrate=0.0*, *dt=0.025*, *v_init=- 75.0*)
Subclass of *NeuronSimTree* where sections are defined so that they are effectively single compartments. Should be created from a *neat.CompartmentTree* using *neat.createReducedCompartmentModel()*

—

createReducedNeuronModel(*ctree*, *fake_c_m=1.0*, *fake_r_a=9.999999999999999e-05*, *method=2*)
Creates a *neat.NeuronCompartmentTree* to simulate reduced compartmentment models from a *neat.CompartmentTree*.

Parameters **ctree** (*neat.CompartmentTree*) – The tree containing the parameters of the reduced compartmental model to be simulated

Returns

Return type *neat.NeuronCompartmentTree*

Notes

The function *ctree.getEquivalentLocs()* can be used to obtain 'fake' locations corresponding to each compartment, which in turn can be used to insert hoc point process at the compartments using the same functions definitions as for as for a morphological *neat.NeuronSimTree*

—

6.3.2 Morphological Trees

Morphology Tree

class MorphTree(*file_n=None*, *types=[1, 3, 4]*)

Subclass of simple tree that implements neuronal morphologies. Reads in trees from '.swc' files (http://neuromorpho.org/).

Neural morphologies are assumed to follow the three-point soma conventions. Internally however, the soma is represented as a sphere. Hence nodes with indices 2 and 3 do not represent anything and are skipped in iterations and getters.

Can also store a simplified version of the original tree, where only nodes are retained that should hold computational parameters - the root, the bifurcation nodes and the leafs at least, although the user can also specify additional nodes. One tree is set as primary by changing the *treetype* attribute (select 'original' for the original morphology and 'computational' for the computational morphology). Lookup operations will often use the primary tree. Using nodes from the other tree for lookup operations is unsafe and should be avoided, it is better to set the proper tree to primary first.

For computational efficiency, it is possible to store sets of locations on the morphology, under user-specified names. These sets are stored as lists of *neat.MorphLoc*, and associated arrays are stored that contain the corresponding node indices of the locations, their x-coordinates, their distances to the soma and their distances to the nearest bifurcation in the 'up'-direction

Parameters

- **file_n** (*str (optional)*) – the file name of the morphology file. Assumed to follow the '.swc' format. Default is `None`, which initialized an empty tree
- **types** (*list of int (optional)*) – The list of node types to be included. As per the '.swc' convention, `1` is soma, `2` is axon, `3` is basal dendrite and `4` apical dendrite. Default is `[1,3,4]`.

Variables

- ***root*** (*neat.MorphNode* instance) – The root of the tree.
- **locs** (dict {str: list of *neat.MorphLoc*}) – Stored sets of locations, key is the user-specified the name of the set of locations. Initialized as empty dict.
- **nids** (*dict {str: np.array of int}*) – Node indices of locations Initialized as empty dict.

- **xs** (*dict {str: np.array of float}*) – x-coordinates of locations Initialized as empty dict.
- **d2s** (*dict {str: np.array of float}*) – distances to soma of locations Initialized as empty dict.
- **d2b** (*dict {str: np.array of float}*) – distances to nearest bifurcation in 'up' direction of locations. Initialized as empty dict.

Read a morphology from an SWC file

MorphTree.readSWCTreeFromFile(file_n[, types])	Non-specific for a "tree data structure" Read and load a morphology from an SWC file and parse it into an *neat.MorphTree* object.
MorphTree.determineSomaType(file_n)	Determine the soma type used in the SWC file.

neat.MorphTree.readSWCTreeFromFile

MorphTree.**readSWCTreeFromFile**(*file_n*, *types=[1, 3, 4]*)

Non-specific for a "tree data structure" Read and load a morphology from an SWC file and parse it into an *neat.MorphTree* object.

On the NeuroMorpho.org website, 5 types of somadescriptions are considered (http://neuromorpho.org/neuroMorpho/SomaFormat.html). The "3-point soma" is the standard and most files are converted to this format during a curation step. *neat* follows this default specification and the *internal structure of `neat` implements the 3-point soma*. Additionally multi-cylinder descriptions with more than three nodes are also supported, but are converted to the standard three point description.

Additionally, the root node of the tree must have `index == 1`, `swc_type == 1` and occur first in the SWC file.

Parameters

- **file_n** (*str*) – name of the file to open
- **types** (*list of ints*) – NeuroMorpho.org segment types to be loaded

Examples

The three point description is

```
1 1 x y   z r -1
1 1 x y-r z r 1
1 1 x y+r z r 1
```

with *x,y,z* the coordinates of the soma center and *r* the soma radius

This is a valid three point desciption

```
# start of file
1 1 45.3625 18.6775 -50.25 10.1267403895 -1
2 1 45.3625 8.55075961052 -50.25 10.1267403895 1
3 1 45.3625 28.8042403895 -50.25 10.1267403895 1
# dendrite nodes
4 3 37.76 12.99 -46.08 0.29 1
5 3 26.7068019951 8.26344199599 -36.9426896493 0.795614809475 4
# ...
```

This is a valid multi-cylinder descirption

```
# start of file
1 1 1066.38 399.67 157.0 4.9215 -1
2 1 1071.3 399.67 157.0 4.9215 1
3 1 1076.22 399.67 157.0 4.9215 2
4 1 1066.5 402.83 157.0 11.494 2
5 1 1062.4 405.5 157.0 15.308 4
6 1 1056.6 410.25 158.0 20.536 5
7 1 1056.6 410.25 158.0 20.536 6
8 1 1070.0 427.75 161.0 2.305 7
# dendrite nodes
9 3 1070.0 427.75 161.0 0.886 8
# ...
```

Raises **ValueError** – If the SWC file is not consistent with the aforementioned conventions

neat.MorphTree.determineSomaType

MorphTree.**determineSomaType**(*file_n*)

Determine the soma type used in the SWC file. This method searches the whole file for soma entries.

Only tbe standard three-point soma type and a multi-cylinder description are supported.

Furthermore, the root node of the tree must have `index == 1`, `swc_type == 1` and occur first in the SWC file.

Parameters **file_n** (*string*) – Name of the file containing the SWC description

Returns **soma_type** – Integer indicating one of the su[pported SWC soma formats. 1: Default three-point soma, 2: multiple cylinder description

Return type int

Raises **ValueError** – If soma type is not supported (less than three nodes have soma)

MorphTree.__getitem__(index[, skip_inds])	Returns the node with given index, if no such node is in the tree, None is returned.
MorphTree.__iter__([node, skip_inds])	Overloaded iterator from parent class that avoids iterating over the nodes with index 2 and 3

neat.MorphTree.__getitem__

MorphTree.**__getitem__**(*index*, *skip_inds=(2, 3)*)

Returns the node with given index, if no such node is in the tree, None is returned.

Parameters **index** (*int*) – the index of the node to be found

Returns: *neat.MorphNode* or None

neat.MorphTree.__iter__

MorphTree.**__iter__**(*node=None*, *skip_inds=(2, 3)*)

Overloaded iterator from parent class that avoids iterating over the nodes with index 2 and 3

Parameters

- **node** (*neat.MorphNode*) – The starting node. Defaults to the root
- **skip_inds** (*tuple of ints*) – Indices of the nodes that are skipped by the iterator. Defaults to `(2,3)`, the nodes that contain extra geometrical information on the soma.

Yields *neat.MorphNode* – Nodes in the tree

Get specific nodes or sets of nodes from the tree.

MorphTree.root	Returns the root of the original or the computational tree, depending on which *treetype* is active.
MorphTree.getNodes([recompute_flag, skip_inds])	Overloads the parent function to allow skipping nodes with certain indices and to return the nodes associated with the corresponding *treetype*.
MorphTree.nodes	Overloads the parent function to allow skipping nodes with certain indices and to return the nodes associated with the corresponding *treetype*.
MorphTree.getLeafs([recompute_flag])	Overloads the *getLeafs* of the parent class to return the leafs in the current *treetype*.
MorphTree.leafs	Overloads the *getLeafs* of the parent class to return the leafs in the current *treetype*.
MorphTree.getNodesInBasalSubtree()	Return the nodes associated with the basal subtree
MorphTree.getNodesInApicalSubtree()	Return the nodes associated with the apical subtree
MorphTree.getNodesInAxonalSubtree()	Return the nodes associated with the apical subtree
MorphTree._convertNodeArgToNodes(node_arg)	Converts a node argument to a list of nodes.

neat.MorphTree.root

property MorphTree.**root**

Returns the root of the original or the computational tree, depending on which *treetype* is active.

neat.MorphTree.getNodes

MorphTree.**getNodes**(*recompute_flag=0*, *skip_inds=(2, 3)*)

Overloads the parent function to allow skipping nodes with certain indices and to return the nodes associated with the corresponding *treetype*.

Parameters

- **recompute_flag** (*bool*) – whether or not to re-evaluate the node list. Defaults to False.
- **skip_inds** (*tuple of ints*) – Indices of the nodes that are skipped by the iterator. Defaults to `(2,3)`, the nodes that contain extra geometrical information on the soma.

Returns

Return type list of *neat.MorphNode*

neat.MorphTree.nodes

property MorphTree.**nodes**

Overloads the parent function to allow skipping nodes with certain indices and to return the nodes associated with the corresponding *treetype*.

Parameters

- **recompute_flag** (*bool*) – whether or not to re-evaluate the node list. Defaults to False.
- **skip_inds** (*tuple of ints*) – Indices of the nodes that are skipped by the iterator. Defaults to `(2,3)`, the nodes that contain extra geometrical information on the soma.

Returns

Return type list of *neat.MorphNode*

neat.MorphTree.getLeafs

MorphTree.**getLeafs**(*recompute_flag=0*)

Overloads the *getLeafs* of the parent class to return the leafs in the current *treetype*.

Parameters **recompute_flag** (*bool*) – Whether to force recomputing the leaf list. Defaults to 0.

neat.MorphTree.leafs

property MorphTree.**leafs**

Overloads the *getLeafs* of the parent class to return the leafs in the current *treetype*.

Parameters **recompute_flag** (*bool*) – Whether to force recomputing the leaf list. Defaults to 0.

neat.MorphTree.getNodesInBasalSubtree

MorphTree.**getNodesInBasalSubtree**()

Return the nodes associated with the basal subtree

Returns List of all nodes in the basal subtree

Return type list of *neat.MorphNode*

neat.MorphTree.getNodesInApicalSubtree

MorphTree.**getNodesInApicalSubtree**()

Return the nodes associated with the apical subtree

Returns List of all nodes in the apical subtree

Return type list of *neat.MorphNode*

neat.MorphTree.getNodesInAxonalSubtree

MorphTree.**getNodesInAxonalSubtree**()
Return the nodes associated with the apical subtree

Returns List of all nodes in the apical subtree

Return type list of *neat.MorphNode*

neat.MorphTree._convertNodeArgToNodes

MorphTree.**_convertNodeArgToNodes**(*node_arg*)
Converts a node argument to a list of nodes. Behaviour depends on the type of argument.

Parameters **node_arg** (None, *neat.MorphNode*, {'apical', 'basal', 'axonal'} or iterable collection of instances of *neat.MorphNode*) –

- *None*: returns all nodes
- *neat.MorphNode*: returns list of nodes in the subtree of the given node
- {'apical', 'basal', 'axonal'}: returns list of nodes in the apical, basal or axonal subtree
- **iterable collection of *neat.MorphNode*: returns the same list of nodes** If an iterable collection of original nodes is given, and the treetype is computational, a reduced list is returned where only the corresponding computational nodes are included. If an iterable collection of computational nodes is given, and the treetype is original, a list of corresponding original nodes is given, but the in between nodes are not added.

Returns

Return type list of *neat.MorphNode*

Relating to the computational tree.

MorphTree.setTreetype(treetype)	Set the active tree
MorphTree.treetype	
MorphTree.readSWCTreeFromFile(file_n[, types])	Non-specific for a "tree data structure" Read and load a morphology from an SWC file and parse it into an *neat.MorphTree* object.
MorphTree.setCompTree([compnodes, ...])	Sets the nodes that contain computational parameters.
MorphTree._evaluateCompCriteria(node[, eps, ...])	Return True if relative difference between node radius and parent node raidus is larger than margin eps, or if the node is the root or bifurcation node.
MorphTree.removeCompTree()	Removes the computational tree

neat.MorphTree.setTreetype

MorphTree.**setTreetype**(*treetype*)
Set the active tree

Parameters **treetype** (*'original' or 'computational'*) – the treetype thas is set to active

neat.MorphTree.treetype

property MorphTree.**treetype**

neat.MorphTree.setCompTree

MorphTree.**setCompTree**(*compnodes=None*, *set_as_primary_tree=False*, *eps=1e-08*)
Sets the nodes that contain computational parameters. This are a priori either bifurcations, leafs, the root or nodes where the neurons' relevant parameters change.

Parameters

- **compnodes** (list of ::class::*MorphNode*) – list of nodes that should be retained in the computational tree. Note that specifying bifurcations, leafs or the root is superfluous, since they are part of the computational tree by default.
- **set_as_primary_tree** (bool (default False)) – if True, sets the computational tree as the primary tree
- **eps** (float (default 1e-8)) – relative margin for parameter change

neat.MorphTree._evaluateCompCriteria

MorphTree.**_evaluateCompCriteria**(*node*, *eps=1e-08*, *rbool=False*)
Return True if relative difference between node radius and parent node raidus is larger than margin eps, or if the node is the root or bifurcation node.

Parameters

- **node** (*neat.MorphNode*) – node that is compared to parent node
- **eps** (float (optional, default 1e-8)) – the margin

Returns

Return type bool

neat.MorphTree.removeCompTree

MorphTree.**removeCompTree**()
Removes the computational tree

Storing locations, interacting with stored locations and distributing locations

MorphTree._convertLocArgToLocs(locarg)	Converts locations argument to list of *neat.MorphLoc*.
MorphTree.storeLocs(*args, **kwargs)	Store locations under a specified name

continues on next page

Table 11 – continued from previous page

MorphTree.addLoc(*args, **kwargs)	Add location to set of locations of given name
MorphTree.clearLocs()	Remove all set of locs stored in the tree
MorphTree.removeLocs(name)	Remove a set of locations of a given name
MorphTree._tryName(name)	Tests if the name is in use.
MorphTree.getLocs(name)	Returns a set of locations of a specified name
MorphTree.getNodeIndices(name)	Returns an array of node indices of locations of a specified name
MorphTree.getXCoords(name)	Returns an array of x-values of locations of a specified name
MorphTree.getLocindsOnNode(name, node)	Returns a list of the indices of locations in the list of a given name that are on a the input node, ordered for increasing x
MorphTree.getLocindsOnNodes(name, node_arg)	Returns a list of the indices of locations in the list of a given name that are on one of the nodes specified in the node list.
MorphTree.getLocindsOnPath(name, node0, node1)	Returns a list of the indices of locations in the list of a given name that are on the given path.
MorphTree.getNearestLocinds(locs, name[, ...])	For each location in the input location list, find the index of the closest location in a set of locations stored under a given name.
MorphTree.getNearestNeighbourLocinds(loc, locarg)	Search nearest neighbours to *loc* in *locarg*.
MorphTree.getLeafLocinds(name[, recompute])	Find the indices in the desire location list that are 'leafs', i.e. locations for which no other location exist that is farther from the root.
MorphTree.distancesToSoma(locarg[, recompute])	Compute the distance of each location in a given set to the soma
MorphTree.distancesToBifurcation(name[, ...])	Compute the distance of each location to the nearest bifurcation in the 'up' direction (towards root)
MorphTree.distributeLocsOnNodes(d2s[, ...])	Distributes locs on a given set of nodes at specified distances from the soma.
MorphTree.distributeLocsUniform(*args, **kwargs)	Distributes locations as uniform as possible, i.e. for a given distance between locations *dx*, locations are distributed equidistantly on each given node in the computational tree and their amount is computed so that the distance in between them is as close to *dx* as possible.
MorphTree.distributeLocsRandom(num[, dx, ...])	Returns a list of input locations randomly distributed on the tree
MorphTree.extendWithBifurcationLocs(loc_arg)	Extends input loc_arg with the intermediate bifurcations.
MorphTree.uniqueLocs(loc_arg[, name])	Gets the unique locations in the provided locs
MorphTree.pathLength(loc1, loc2[, ...])	Find the length of the direct path between loc1 and loc2

neat.MorphTree._convertLocArgToLocs

MorphTree.**_convertLocArgToLocs**(*locarg*)
Converts locations argument to list of *neat.MorphLoc*.

Parameters **locarg** (list of dictionaries, tuples or *neat.MorphLoc*, or string) –

- If list, entries should be valid arguments to initialize a *neat.MorphLoc*
- If string, should be the name of a list of locations stored in *self*

Returns List of locations, each referencing the current tree

Return type list of *neat.MorphLoc*

neat.MorphTree.storeLocs

MorphTree.**storeLocs**(**args*, ***kwargs*)
Store locations under a specified name

Parameters

- **locs** (list of dicts, tuples or *neat.MorphLoc*) – the locations to be stored
- **name** (*string*) – name under which these locations are stored
- **warn** (bool (default `True`)) – raise a *UserWarning* if two or more locations in *locs* refer to the soma. Choose `False` if this is desired to remove the warning.

neat.MorphTree.addLoc

MorphTree.**addLoc**(**args*, ***kwargs*)
Add location to set of locations of given name

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – the location to be added
- **name** (*str*) – the name of the set of locations to which the location is added

neat.MorphTree.clearLocs

MorphTree.**clearLocs**()
Remove all set of locs stored in the tree

neat.MorphTree.removeLocs

MorphTree.**removeLocs**(*name*)
Remove a set of locations of a given name

Parameters **name** (*string*) – name under which the desired list of locations is stored

neat.MorphTree._tryName

MorphTree.**_tryName**(*name*)
Tests if the name is in use. Raises a KeyError when it is not in use and prints a list of possible names

Parameters **name** (*string*) – name of the desired list of locations

Raises **KeyError** – If 'name' does not refer to a set of locations in use

neat.MorphTree.getLocs

MorphTree.**getLocs**(*name*)
Returns a set of locations of a specified name

Parameters **name** (*string*) – name under which the desired list of locations is stored

Returns

Return type list of *neat.MorphLoc*

neat.MorphTree.getNodeIndices

MorphTree.**getNodeIndices**(*name*)
Returns an array of node indices of locations of a specified name

Parameters **name** (*string*) – name under which the desired list of locations is stored

Returns

Return type numpy.array of ints

neat.MorphTree.getXCoords

MorphTree.**getXCoords**(*name*)
Returns an array of x-values of locations of a specified name

Parameters **name** (*string*) – name under which the desired list of locations is stored

neat.MorphTree.getLocindsOnNode

MorphTree.**getLocindsOnNode**(*name*, *node*)
Returns a list of the indices of locations in the list of a given name that are on a the input node, ordered for increasing x

Parameters

- **name** (*string*) – which list of locations to consider
- **node** (*neat.MorphNode*) – the node to consider. Should be part of the original tree

Returns indices of locations on the path

Return type list of ints

neat.MorphTree.getLocindsOnNodes

MorphTree.**getLocindsOnNodes**(*name*, *node_arg*)

Returns a list of the indices of locations in the list of a given name that are on one of the nodes specified in the node list. Within each node, locations are ordered for increasing x

Parameters

- **name** (*string*) – which list of locations to consider
- **node_arg** – see documentation of *MorphTree._convertNodeArgToNodes*

Returns indices of locations on the path

Return type list of ints

neat.MorphTree.getLocindsOnPath

MorphTree.**getLocindsOnPath**(*name*, *node0*, *node1*, *xstart=0.0*, *xstop=1.0*)

Returns a list of the indices of locations in the list of a given name that are on the given path. The path is taken to start at the input x-start coordinate of the first node in the list and to stop at the given x-stop coordinate of the last node in the list

Parameters

- **name** (*string*) – which list of locations to consider
- **node0** (*SNode*) – start node of path
- **node1** (*SNode*) – stop node of path
- **xstart** (float (in `[0,1]`)) – starting coordinate on *node0*
- **xstop** (float (in `[0,1]`)) – stopping coordinate on *node1*

Returns Indices of locations on the path. If path is empty, an empty array is returned.

Return type list of ints

neat.MorphTree.getNearestLocinds

MorphTree.**getNearestLocinds**(*locs*, *name*, *direction=0*, *check_siblings=True*, *pprint=False*)

For each location in the input location list, find the index of the closest location in a set of locations stored under a given name. The search can go in the either go in the up or down direction or in both directions.

Parameters

- **locs** (list of dicts, tuples or *neat.MorphLoc*) – the locations for which the nearest location index has to be found
- **name** (*string*) – name under which the reference list is stored
- **direction** (*int*) – flag to indicate whether to search in both directions (0), only in the up direction (1) or in the down direction (2).

Returns **loc_indices** – indices of the locations closest to the given locs

Return type list of ints

neat.MorphTree.getNearestNeighbourLocinds

MorphTree.**getNearestNeighbourLocinds**(*loc*, *locarg*)

Search nearest neighbours to *loc* in *locarg*.

Parameters

- **loc** (tuple, dict or *neat.MorphLoc*) – The locations for which nearest neighbours have to be found
- **locarg** (*str or list of locs*) – See documentation of *MorphTree._parseLocArg*, the set of locations within which to look for nearest neighbours

Returns Indices of nearest neighbours of *loc* in *locarg*

Return type list of ints

neat.MorphTree.getLeafLocinds

MorphTree.**getLeafLocinds**(*name*, *recompute=False*)

Find the indices in the desire location list that are 'leafs', i.e. locations for which no other location exist that is farther from the root

Parameters

- **name** (*string*) – name of the desired set of locations
- **recompute** (bool (optional, default False)) – whether or not to force recomputing the distances

Returns the indices of the 'leaf' locations

Return type list of inds

neat.MorphTree.distancesToSoma

MorphTree.**distancesToSoma**(*locarg*, *recompute=False*)

Compute the distance of each location in a given set to the soma

Parameters **locarg** (*list of locations or string*) – if list of locations, specifies the locations, if str, specifies the name under which the set of location is stored that should be used to create the new tree

Returns

- *np.array of float* – the distances to the soma of the corresponding locations
- **recompute** (*bool (optional)*) – whether or not to force recomputing the distances

neat.MorphTree.distancesToBifurcation

MorphTree.**distancesToBifurcation**(*name*, *recompute=False*)

Compute the distance of each location to the nearest bifurcation in the 'up' direction (towards root)

Parameters

- **name** (*str*) – name of the set of locations
- **recompute** (bool (optional, default False)) – whether or not to force recomputing the distances

Returns the distances to the nearest bifurcation of the corresponding locations

Return type np.array of floats

neat.MorphTree.distributeLocsOnNodes

MorphTree.**distributeLocsOnNodes**(*d2s*, *node_arg=None*, *name='dont save'*)

Distributes locs on a given set of nodes at specified distances from the soma. If the specified distances are on the specified nodes, the list of locations will be empty. The locations are stored if the name is set to be something other than 'dont save'. On each node, locations are ordered from low to high x-values.

Parameters

- **d2s** (*numpy.array of floats*) – the distances from the soma at which to put the locations (micron)
- **node_arg** – see documentation of *MorphTree._convertNodeArgToNodes*
- **name** (*string*) – the name under which the locations are stored. Defaults to 'dont save' which means the locations are not stored

Returns the list of locations

Return type list of *neat.MorphLoc*

neat.MorphTree.distributeLocsUniform

MorphTree.**distributeLocsUniform**(**args*, ***kwargs*)

Distributes locations as uniform as possible, i.e. for a given distance between locations *dx*, locations are distributed equidistantly on each given node in the computational tree and their amount is computed so that the distance in between them is as close to *dx* as possible. Depth-first ordering.

Parameters

- **dx** (*float (> 0)*) – target distance in micron between the locations
- **node_arg** – see documentation of *MorphTree._convertNodeArgToNodes*
- **add_bifurcations** (*bool*) – whether to ensure that all bifurcation nodes are added
- **name** (*string*) – the name under which the locations are stored. Defaults to 'dont save' which means the locations are not stored

Returns the list of locations

Return type list of *neat.MorphLoc*

neat.MorphTree.distributeLocsRandom

MorphTree.**distributeLocsRandom**(*num*, *dx=0.001*, *node_arg=None*, *add_soma=True*, *name='dont save'*, *seed=None*)

Returns a list of input locations randomly distributed on the tree

Parameters

- **num** (*int*) – number of inputs
- **dx** (*float (optional)*) – minimal or given distance between input locations (micron)
- **(optional)** (*node_arg*) – see documentation of *MorphTree._convertNodeArgToNodes*
- **add_soma** (*bool (optional)*) – whether or not to include the possibility of adding locations on the soma
- **name** (*string (optional)*) – the name under which the locations are stored. Defaults to 'dont save' which means the locations are not stored
- **seed** (*int (optiona)*) – Seed for numpy random number generator

Returns the locations

Return type list of *neat.MorphLoc*

neat.MorphTree.extendWithBifurcationLocs

MorphTree.**extendWithBifurcationLocs**(*loc_arg*, *name='dont save'*)

Extends input loc_arg with the intermediate bifurcations. They are appended to the end of the list

Parameters

- **loc_arg** (list of *neat.MorphLoc* or string) – the locations
- **name** (*string (optional)*) – The name under which the list of bifurcation locs will be stored. Defaults to 'dont save' which means they are not stored.

Returns the bifurcation locs

Return type list of *neat.MorphLoc*

neat.MorphTree.uniqueLocs

MorphTree.**uniqueLocs**(*loc_arg*, *name='dont save'*)

Gets the unique locations in the provided locs

Parameters

- **loc_arg** (list of *neat.MorphLoc* or string) – the locations
- **name** (*string (optional)*) – The name under which the list of bifurcation locs will be stored. Defaults to 'dont save' which means they are not stored.

Returns the bifurcation locs

Return type list of *neat.MorphLoc*

neat.MorphTree.pathLength

MorphTree.**pathLength**(*loc1*, *loc2*, *compute_radius=0*)

Find the length of the direct path between loc1 and loc2

Parameters

- **loc1** (dict, tuple or *neat.MorphLoc*) – one location
- **loc2** (dict, tuple or *neat.MorphLoc*) – other location
- **compute_radius** (*bool*) – if True, also computes the average weighted radius of the path

Returns

L: float length of path, in micron

R: float weighted average radius of path, in micron

Return type L, R (optional)

Plotting on a 1D axis.

MorphTree.makeXAxis([dx, node_arg, loc_arg])	Create a set of locs suitable for serving as the x-axis for 1D plotting.
MorphTree.setNodeColors([startnode])	Set the color code for the nodes for 1D plotting
MorphTree.getXValues(locs)	Get the corresponding location on the x-axis of the input locations
MorphTree.plot1D(ax, parr, *args, **kwargs)	Plot an array where each element corresponds to the matching location on the x-axis with a depth-first ordering on a 1D plot
MorphTree.plotTrueD2S(ax, parr[, cmap])	Plot an array where each element corresponds to the matching location in the x-axis location list.
MorphTree.colorXAxis(ax, cmap[, ...])	Color the x-axis of a plot according to the morphology.

neat.MorphTree.makeXAxis

MorphTree.**makeXAxis**(*dx=10.0*, *node_arg=None*, *loc_arg=None*)

Create a set of locs suitable for serving as the x-axis for 1D plotting. The neurons is put on a 1D axis with a depth-first ordering.

Parameters

- **dx** (*float*) – target separation between the plot points (micron)
- **node_arg** – see documentation of *MorphTree._convertNodeArgToNodes* The nodes on which the locations for the x-axis are distributed. When this is given as a list of nodes, assumes a depth first ordering.
- **loc_arg** (*list of locs or string*) – if list of locs, these locs will be used as x-axis, if string, name of set of locs on the morphology that will be used as x-axis

neat.MorphTree.setNodeColors

MorphTree.**setNodeColors**(*startnode=None*)

Set the color code for the nodes for 1D plotting

Parameters **node** (int or *neat.MorphNode*) – index of the node or node whose subtree will be colored. Defaults to the root

neat.MorphTree.getXValues

MorphTree.**getXValues**(*locs*)

Get the corresponding location on the x-axis of the input locations

Parameters **locs** (list of tuples, dicts or *neat.MorphLoc*) – list of the locations

neat.MorphTree.plot1D

MorphTree.**plot1D**(*ax*, *parr*, **args*, ***kwargs*)

Plot an array where each element corresponds to the matching location on the x-axis with a depth-first ordering on a 1D plot

Parameters

- **ax** (*matplotlib.axes.Axes* instance) – the ax object on which the plot will be made
- **parr** (*numpy.array of floats*) – the array that will be plotted
- **args** – arguments for *matplotlib.pyplot.plot*
- **kwargs** – arguments for *matplotlib.pyplot.plot*

Returns **lines** – the line segments corresponding to the value of the plotted array in each branch

Return type list of *matplotlib.lines.Line2D* instances

Raises **AssertionError** – When the number of elements in the data array in not equal to the number of elements on the x-axis

neat.MorphTree.plotTrueD2S

MorphTree.**plotTrueD2S**(*ax*, *parr*, *cmap=None*, ***kwargs*)

Plot an array where each element corresponds to the matching location in the x-axis location list. Now all locations are plotted at their true distance from the soma.

Parameters

- **ax** (*matplotlib.axes.Axes* instance) – the ax object on which the plot will be made
- **parr** (*numpy.array of floats*) – the array that will be plotted
- **cmap** (*matplotlib.colors.Colormap* instance) – If provided, the lines will be colored according to the branch to which they belong, in colors specified by the colormap
- **kwargs** – keyword arguments for *matplotlib.pyplot.plot*

Returns

- *lines*

- **lines** (list of *matplotlib.lines.Line2D*) – the line segments corresponding to the value of the plotted array in each branch

Raises **AssertionError** – When the number of elements in the data array in not equal to the number of elements on the x-axis

neat.MorphTree.colorXAxis

MorphTree.**colorXAxis**(*ax*, *cmap*, *addScalebar=1*, *borderpad=- 1.8*)

Color the x-axis of a plot according to the morphology.

!!! Has to be called after all lines are plotted !!!

Furthermor, node colors have to be set first. This can be done with *MorphTree.setNodeColors()* or manually by adding a 'color' entry to the `MorphNode.content` dictionary

Parameters

- **ax** (*matplotlib.axes.Axes* instance) – the ax object of which the x-axis will be colored
- **cmap** (*matplotlib.colors.Colormap* instance) – Colormap that determines the color of each branch
- **sizex** (*float*) – Size of scalebar (in micron). If set to None, no scalebar is plotted.
- **borderpad** (*float*) – Borderpad of scalebar

Plotting the morphology in 2D.

MorphTree.plot2DMorphology(ax[, node_arg, ...])	Plot the morphology projected on the x,y-plane
MorphTree.plotMorphologyInteractive([...])	Show the morphology either in 3d or projected on the x,y-plane.

neat.MorphTree.plot2DMorphology

MorphTree.**plot2DMorphology**(*ax*, *node_arg=None*, *cs=None*, *cminmax=None*, *cmap=None*, *use_radius=1*, *draw_soma_circle=1*, *plotargs={}*, *textargs={}*, *marklocs=[]*, *locargs={}*, *marklabels={}*, *labelargs={}*, *cb_draw=0*, *cb_orientation='vertical'*, *cb_label=''*, *sb_draw=1*, *sb_scale=100*, *sb_width=5.0*, *set_lims=True*, *lims_margin=0.1*)

Plot the morphology projected on the x,y-plane

Parameters

- **ax** (*matplotlib.axes.Axes* instance) – the ax object on which the plot will be drawn
- **node_arg** – see documentation of *MorphTree._convertNodeArgToNodes*
- **cs** (*dict {int: float}, None or 'x_color'*) – If dict, node indices are keys and the float value will correspond to the plotted color. If None, the color of the tree will be the one specified in `plotargs`. Note that the dict does not have to contain all node indices. The ones that are not featured in the dict are plot in the color specified in `plotargs`. If 'node_color', colors will be those stored on the nodes. Note that choosing this option when there are nodes without 'color' as an entry in `node.content` will result in an error. Node colors can be set with *MorphTree.setNodeColor()*`
- **cminmax** (*(float, float) or None (default)*) – The min and max values of

the color scale (if cs is provided). If None, the min and max values of cs are used.

- **cmap** (*matplotlib.colors.Colormap* instance) – colormap fram which colors in `cs` are taken
- **use_radius** (*bool*) – If `True`, uses the swc radius for the width of the line segments
- **draw_soma_circle** (*bool*) – If `True`, draws the soma as a circle, otherwise doesn't draw soma
- **plotargs** (*dict*) – *kwargs* for *matplotlib.pyplot.plot*. 'c'- or 'color'- argument will be overwritten when cs is defined. 'lw'- or 'linewidth' argument will be multiplied with the swc radius of the node if *use_radius* is `True`.
- **textargs** (*dict*) – text properties for various labels in the plot
- **marklocs** (list of tuples, dicts or instances of *neat.MorphLoc*) – Location that will be plotted on the morphology
- **locargs** (*dict or list of dict*) – *kwargs* for *matplotlib.pyplot.plot* for the location. Use only point markers and no lines! When it is a single dict all location will have the same marker. When it is a list it should have the same length as *marklocs*.
- **marklabels** (*dict {int: string}*) – Keys are indices of locations in *marklocs*, values are strings that are used to annotate the corresponding locations
- **labelargs** (*dict*) – text properties for the location annotation
- **cb_draw** (*bool*) – Whether or not to draw a *matplotlib.pyplot.colorbar()* instance.
- **cb_orientation** (*string, 'vertical' or 'horizontal'*) – The colorbars' orientation
- **cb_label** (*string*) – The label of the colorbar
- **sb_draw** (*bool*) – Whether or not to draw a scale bar
- **sb_scale** (*float*) – Lenght of the scale bar (micron)
- **sb_width** (*float*) – Width of the scale bar
- **set_lims** (bool (optional, default `True`)) – set `ax` limits based on the morphology
- **lims_margin** (*float*) – the margin, as fraction of total width and height of tree, at which the limits are placed

neat.MorphTree.plotMorphologyInteractive

MorphTree.**plotMorphologyInteractive**(*node_arg=None*, *use_radius=1*, *draw_soma_circle=1*, *plotargs={'c': 'k', 'lw': 1.0}*, *project3d=False*)

Show the morphology either in 3d or projected on the x,y-plane. When a line segment is clicked, the associated node is printed.

Parameters

- **ax** (*matplotlib.axes.Axes* instance) – the ax object on which the plot will be drawn
- **node_arg** – see documentation of *MorphTree._convertNodeArgToNodes*
- **use_radius** (*bool*) – If `True`, uses the swc radius for the width of the line segments
- **draw_soma_circle** (*bool*) – If `True`, draws the soma as a circle, otherwise doesn't draw soma

Creating new trees from the existing tree.

MorphTree.createNewTree(*args, **kwargs)	Creates a new tree where the locs of a given 'name' are now the nodes.
MorphTree.createCompartmentTree(*args, **kwargs)	Creates a new compartment tree where the provided set of locations correspond to the nodes.
MorphTree.__copy__([new_tree])	Fill the `new_tree` with it's corresponding nodes in the same structure as `self`, and copies all node variables that both tree classes have in common

neat.MorphTree.createNewTree

MorphTree.**createNewTree**(**args*, ***kwargs*)

Creates a new tree where the locs of a given 'name' are now the nodes. Distance relations between locations are maintained (note that this relation is stored in *L* attribute of *neat.MorphNode*, using the *p3d* attribute containing the 3d coordinates does not maintain distances)

Parameters

- **name** (*string*) – the name under which the locations are stored that should be used to create the new tree
- **fake_soma** (bool (default *False*)) – if *True*, finds the common root of the set of locations and uses that as the soma of the new tree. If *False*, the real soma is used.
- **store_loc_inds** (bool (default *False*)) – store the index of each location in the *content* attribute of the new node (under the key 'loc ind')

Returns The new tree.

Return type *neat.MorphTree*

neat.MorphTree.createCompartmentTree

MorphTree.**createCompartmentTree**(**args*, ***kwargs*)

Creates a new compartment tree where the provided set of locations correspond to the nodes.

Parameters **locarg** (*list of locations or str*) – if list of locations, specifies the locations, if str, specifies the name under which the set of location is stored that should be used to create the new tree

Returns The new tree.

Return type *neat.MorphTree*

neat.MorphTree.__copy__

MorphTree.**__copy__**(*new_tree=None*)

Fill the `new_tree` with it's corresponding nodes in the same structure as `self`, and copies all node variables that both tree classes have in common

Parameters **new_tree** (*STree* or derived class (default is `None`)) – the tree class in which the `self` is copied. If `None`, returns a copy of `self`.

Returns

Return type The new tree instance

class MorphNode(*index*, *p3d=None*)

Node associated with *neat.MorphTree*. Stores the geometrical information associated with a point on the tree morphology

Variables

- **xyz** (*numpy.array of floats*) – The xyz-coordinates associated with the node (um)
- **R** (*float*) – The radius of the node (um)
- **swc_type** (*int*) – The type of node, according to the .swc file format convention: 1 is dendrites, 2 is axon, 3 is basal dendrite and 4 is apical dendrite.
- **L** (*float*) – The length of the node (um)

MorphNode.setP3D(xyz, R, swc_type)	Set the 3d parameters of the node
MorphNode.child_nodes	Get the *child_nodes* of this node.

neat.MorphNode.setP3D

MorphNode.**setP3D**(*xyz*, *R*, *swc_type*)

Set the 3d parameters of the node

Parameters

- **xyz** (*np.array*) – 3D location (um)
- **R** (*float*) – Radius of the segment (um)
- **swc_type** (*int*) – Type asscoiated with the segment according to SWC standards

neat.MorphNode.child_nodes

property MorphNode.**child_nodes**

Get the *child_nodes* of this node. Indices 2 and 3 are skipped by default (3-point soma convention)

Parameters **skip_inds** (*list or tuple of ints*) – Node indices of child nodes that are not added to the returned list

Returns The child nodes

Return type list of *neat.MorphNode*

class MorphLoc(*loc*, *reftree*, *set_as_comploc=False*)

Stores a location on the morphology. The location is initialized starting from a node and x-value on the real morphology. The location is also be stored in the coordinates of the computational morphology. To toggle between coordinates, the class stores a reference to the morphology tree on which the location is defined, and returns either the original coordinate or the coordinate on the computational tree, depending on which tree is active.

Initialized based on either a tuple or a dict where one entry specifies the node index and the other entry the x-coordinate specifying the location between parent node (x=0) or the node indicated by the index (x=1), or on a *neat.MorphLoc*.

Parameters

- **loc** (tuple or dict or *neat.MorphLoc*) – if tuple: (node index, x-value) if dict: {'node': node index, 'x': x-value}

- **reftree** (*neat.MorphTree*) –
- **set_as_comploc** (*bool*) – if True, assumes the paremeters provided in *loc* are coordinates on the computational tree. Doing this while no computational tree has been initialized in *reftree* will result in an error. Defaults to False

Raises **ValueError** – If x-coordinate of location is not in `[0,1]`

Physiology Tree

class PhysTree(*file_n=None*, *types=[1, 3, 4]*)

Adds physiological parameters to *neat.MorphTree* and convenience functions to set them across the morphology. Initialized in the same way as *neat.MorphTree*

Variables **channel_storage** (dict {str: *neat.IonChannel*}) – Stores the user defined ion channels present in the tree

PhysTree.asPassiveMembrane(*args, **kwargs)	Makes the membrane act as a passive membrane (for the nodes in `node_arg`), channels are assumed to add a conductance of g_max * p_open to the membrane conductance, where p_open for each node is evaluated at the equilibrium potential stored in that node
PhysTree.setEEq(*args, **kwargs)	Set the equilibrium potentials throughout the tree
PhysTree.setPhysiology(*args, **kwargs)	Set specifice membrane capacitance, axial resistance and (optionally) static point-like shunt conductances in the tree.
PhysTree.setLeakCurrent(*args, **kwargs)	Set the parameters of the leak current.
PhysTree.addCurrent(*args, **kwargs)	Adds a channel to the morphology.
PhysTree.getChannelsInTree(*args, **kwargs)	Returns list of strings of all channel names in the tree
PhysTree.fitLeakCurrent(*args, **kwargs)	Fits the leak current to fix equilibrium potential and membrane time- scale.
PhysTree._evaluateCompCriteria(node[, eps, ...])	Return `True` if relative difference in any physiological parameters between node and child node is larger than margin `eps`.

neat.PhysTree.asPassiveMembrane

PhysTree.**asPassiveMembrane**(**args*, ***kwargs*)

Makes the membrane act as a passive membrane (for the nodes in `node_arg`), channels are assumed to add a conductance of g_max * p_open to the membrane conductance, where p_open for each node is evaluated at the equilibrium potential stored in that node

Parameters **node_arg** (*optional*) – see documentation of *MorphTree._convertNodeArgToNodes()*. Defaults to None. The nodes for which the membrane is set to passive

neat.PhysTree.setEEq

PhysTree.**setEEq**(**args*, ***kwargs*)

Set the equilibrium potentials throughout the tree

Parameters **e_eq_distr** (float, dict or `float -> float()`) – The equilibrium potentials [mV]

neat.PhysTree.setPhysiology

PhysTree.**setPhysiology**(**args*, ***kwargs*)

Set specifice membrane capacitance, axial resistance and (optionally) static point-like shunt conductances in the tree. Capacitance is stored at each node as the attribute 'c_m' (uF/cm2) and axial resistance as the attribute 'r_a' (MOhm*cm)

Parameters

- **c_m_distr** (float, dict or `float -> float()`) – specific membrance capacitance
- **r_a_distr** (float, dict or `float -> float()`) – axial resistance
- **g_s_distr** (float, dict, `float -> float()` or None (optional, default) – is *None*) point like shunt conductances (placed at *(node.index, 1.)* for the nodes in `node_arg`). By default no shunt conductances are added
- **node_arg** (*optional*) – see documentation of *MorphTree._convertNodeArgToNodes()*. Defaults to None

neat.PhysTree.setLeakCurrent

PhysTree.**setLeakCurrent**(**args*, ***kwargs*)

Set the parameters of the leak current. At each node, leak is stored under the attribute *node.currents['L']* at a tuple *(g_l, e_l)* with *g_l* the conductance [uS/cm^2] and *e_l* the reversal [mV]

g_l_distr: float, dict or `float -> float()` If float, the leak conductance is set to this value for all the nodes specified in *node_arg*. If it is a function, the input must specify the distance from the soma (micron) and the output the leak conductance [uS/cm^2] at that distance. If it is a dict, keys are the node indices and values the ion leak conductances [uS/cm^2].

e_l_distr: float, dict or `float -> float()` If float, the reversal [mV] is set to this value for all the nodes specified in *node_arg*. If it is a function, the input must specify the distance from the soma [um] and the output the reversal at that distance. If it is a dict, keys are the node indices and values the ion reversals.

node_arg: optional see documentation of *MorphTree._convertNodeArgToNodes()*. Defaults to None

neat.PhysTree.addCurrent

PhysTree.**addCurrent**(**args*, ***kwargs*)

Adds a channel to the morphology. At each node, the channel is stored under the attribute *node.currents[channel.__class__.__name__]* as a tuple *(g_max, e_rev)* with *g_max* the maximal conductance [uS/cm^2] and *e_rev* the reversal [mV]

Parameters

- **channel_name** (*IonChannel*) – The ion channel
- **g_max_distr** (float, dict or `float -> float()`) – If float, the maximal conductance is set to this value for all the nodes specified in *node_arg*. If it is a function, the input must specify the distance from the soma (micron) and the output the ion channel density (uS/cm^2) at that distance. If it is a dict, keys are the node indices and values the ion channel densities (uS/cm^2).
- **e_rev_distr** (float, dict or `float -> float()`) – If float, the reversal (mV) is set to this value for all the nodes specified in *node_arg*. If it is a function, the input must specify the distance from the soma (micron) and the output the reversal at that distance. If it is a dict, keys are the node indices and values the ion reversals.
- **node_arg** (*optional*) – see documentation of *MorphTree._convertNodeArgToNodes()*. Defaults to None

neat.PhysTree.getChannelsInTree

PhysTree.**getChannelsInTree**(**args*, ***kwargs*)

Returns list of strings of all channel names in the tree

Returns the channel names

Return type list of string

neat.PhysTree.fitLeakCurrent

PhysTree.**fitLeakCurrent**(**args*, ***kwargs*)

Fits the leak current to fix equilibrium potential and membrane time- scale.

!!! Should only be called after all ion channels have been added !!!

Parameters

- **e_eq_target_distr** (float, dict or `float -> float()`) – The target reversal potential (mV). If float, the target reversal is set to this value for all the nodes specified in *node_arg*. If it is a function, the input must specify the distance from the soma (um) and the output the target reversal at that distance. If it is a dict, keys are the node indices and values the target reversals
- **tau_m_target_distr** (float, dict or `float -> float()`) – The target membrane time-scale (ms). If float, the target time-scale is set to this value for all the nodes specified in *node_arg*. If it is a function, the input must specify the distance from the soma (um) and the output the target time-scale at that distance. If it is a dict, keys are the node indices and values the target time-scales
- **node_arg** – see documentation of *MorphTree._convertNodeArgToNodes()*. Defaults to None

neat.PhysTree._evaluateCompCriteria

PhysTree.**_evaluateCompCriteria**(*node*, *eps=1e-08*, *rbool=False*)

Return `True` if relative difference in any physiological parameters between node and child node is larger than margin `eps`.

Overrides the *MorphTree._evaluateCompCriteria()* function called by *MorphTree.setCompTree()*.

Parameters

- **node** (::class::*MorphNode*) – node that is compared to parent node
- **eps** (float (optional, default `1e-8`)) – the margin

Returns

Return type bool

class PhysNode(*index*, *p3d=None*, *c_m=1.0*, *r_a=9.999999999999999e-05*, *g_shunt=0.0*, *e_eq=- 75.0*)

Node associated with *neat.PhysTree*. Stores the physiological parameters of the cylindrical segment connecting this node with its parent node

Variables

- **currents** (*dict {str: [float,float]}*) – dict with as keys the channel names and as values lists of length two containing as first entry the channels' conductance density (uS/cm^2) and as second element the channels reversal (mV) (i.e.: {name: [g_max (uS/cm^2), e_rev (mV)]}) For the leak conductance, the corresponding key is 'L'
- **concmechs** (*dict*) – dict containing concentration mechanisms present in the segment
- **c_m** (*float*) – The sement's specific membrane capacitance (uF/cm^2)
- **r_a** (*float*) – The segment's axial resistance (MOhm*cm)
- **g_shunt** (*float*) – Point-like shunt conductance located at x=1 (uS)
- **e_eq** (*float*) – Segment's equilibrium potential

Separation of Variables Tree

class SOVTree(*file_n=None*, *types=[1, 3, 4]*)

Class that computes the separation of variables time scales and spatial mode functions for a given morphology and electrical parameter set. Employs the algorithm by (Major, 1994). This three defines a special *neat.SomaSOVNode* on as a derived class from *neat.SOVNode* as some functions required for SOV calculation are different and thus overwritten.

The SOV calculation proceeds on the computational tree (see docstring of *neat.MorphNode*). Thus it makes no sense to look for sov quantities in the original tree.

SOVTree.calcSOVEquations(*args, **kwargs)	Calculate the timescales and spatial functions of the separation of variables approach, using the algorithm by (Major, 1993).
SOVTree.getModeImportance([locarg, ...])	Gives the overal importance of the SOV modes for a certain set of locations
SOVTree.getImportantModes([locarg, ...])	Returns the most importand eigenmodes (those whose importance is above the threshold defined by *eps*)
SOVTree.calcImpedanceMatrix([locarg, ...])	Compute the impedance matrix for a set of locations
SOVTree.constructNET([dz, dx, eps, ...])	Construct a Neural Evaluation Tree (NET) for this cell

continues on next page

Table 17 – continued from previous page

SOVTree.computeLinTerms(net[, sov_data, eps])	Construct linear terms for *net* so that transfer impedance to soma is exactly matched

neat.SOVTree.calcSOVEquations

SOVTree.**calcSOVEquations**(**args*, ***kwargs*)

Calculate the timescales and spatial functions of the separation of variables approach, using the algorithm by (Major, 1993).

The (reciprocals) of the timescales (i.e. the roots of the transcendental equation) are stored in the somanode. The spatial factors are stored in each (computational) node.

Parameters **maxspace_freq** (*float (default is 500)*) – roughly corresponds to the maximal spatial frequency of the smallest time-scale mode

neat.SOVTree.getModeImportance

SOVTree.**getModeImportance**(*locarg=None*, *sov_data=None*, *importance_type='simple'*)

Gives the overal importance of the SOV modes for a certain set of locations

Parameters

- **locarg** (*None or list of locations*) –
- **sov_data** (*None or tuple of mode matrices*) – One of the keyword arguments `locarg` or `sov_data` must not be `None`. If `locarg` is not `None`, the importance is evaluated at these locations (see `neat.MorphTree._parseLocArg()`). If `sov_data` is not `None`, it is a tuple of a vector of the reciprocals of the mode timescales and a matrix with the corresponding spatial mode functions.
- **importance_type** (*string ('relative' or 'absolute')*) – when 'absolute', returns an absolute measure of the importance, when 'relative', normalizes so that maximum importance is one. Defaults to 'relative'.

Returns the importances associated with each mode for the provided set of locations

Return type np.ndarray (ndim = 1)

neat.SOVTree.getImportantModes

SOVTree.**getImportantModes**(*locarg=None*, *sov_data=None*, *eps=0.0001*, *sort_type='timescale'*, *return_importance=False*)

Returns the most importand eigenmodes (those whose importance is above the threshold defined by *eps*)

Parameters

- **locarg** (*None or list of locations*) –
- **sov_data** (*None or tuple of mode matrices*) – One of the keyword arguments `locarg` or `sov_data` must not be `None`. If `locarg` is not `None`, the importance is evaluated at these locations (see `neat.MorphTree._parseLocArg()`). If `sov_data` is not `None`, it is a tuple of a vector of the reciprocals of the mode timescales and a matrix with the corresponding spatial mode functions.
- **eps** (*float*) – the cutoff threshold in relative importance below which modes are truncated

- **sort_type** (*string ('timescale' or 'importance')*) – specifies in which order the modes are returned. If 'timescale', modes are sorted in order of decreasing time-scale, if 'importance', modes are sorted in order of decreasing importance.
- **return_importance** (*bool*) – if `True`, returns the importance metric associated with each mode

Returns

- **alphas** (*np.ndarray of complex (ndim = 1)*) – the reciprocals of mode time-scales `[kHz]`
- **gammas** (*np.ndarray of complex (ndim = 2)*) – the spatial function associated with each mode, evaluated at each locations. Dimension 0 is number of modes and dimension 1 number of locations
- **importance** (np.ndarray (*shape* matches *alphas*, only if *return_importance* is `True`)) – value of importance metric for each mode

neat.SOVTree.calcImpedanceMatrix

`SOVTree.`**calcImpedanceMatrix**(*locarg=None*, *sov_data=None*, *name=None*, *eps=0.0001*, *mem_limit=500*, *freqs=None*)

Compute the impedance matrix for a set of locations

Parameters

- **locarg** (*None or list of locations*) –
- **sov_data** (*None or tuple of mode matrices*) – One of the keyword arguments `locarg` or `sov_data` must not be `None`. If `locarg` is not `None`, the importance is evaluated at these locations (see `neat.MorphTree._parseLocArg()`). If `sov_data` is not `None`, it is a tuple of a vector of the reciprocals of the mode timescales and a matrix with the corresponding spatial mode functions.
- **eps** (*float*) – the cutoff threshold in relative importance below which modes are truncated
- **mem_limit** (*int*) – parameter governs whether the fast (but memory intense) method or the slow method is used
- **freqs** (*np.ndarray of complex or None (default)*) – if `None`, returns the steady state impedance matrix, if a array of complex numbers, returns the impedance matrix for each Fourrier frequency in the array

Returns the impedance matrix, steady state if *freqs* is `None`, the frequency dependent impedance matrix if *freqs* is given, with the frequency dependence at the first dimension `[MOhm ]`

Return type np.ndarray of floats (ndim = 2 or 3)

neat.SOVTree.constructNET

`SOVTree.`**constructNET**(*dz=50.0*, *dx=10.0*, *eps=0.0001*, *use_hist=False*, *add_lin_terms=True*, *improve_input_impedance=False*, *pprint=False*)

Construct a Neural Evaluation Tree (NET) for this cell

Parameters

- **dz** (*float*) – the impedance step for the NET model derivation
- **dx** (*float*) – the distance step to evaluate the impedance matrix
- **eps** (*float*) – the cutoff threshold in relative importance below which modes are truncated

- **use_hist** (*bool*) – whether or not to use histogram segmentations to find well separated parts of the dendritic tree (such ass apical tree)
- **add_lin_terms** – take into account that the optained NET will be used in conjunction with linear terms

Returns The neural evaluation tree (Wybo et al., 2019) associated with the morphology.

Return type *neat.NETree*

neat.SOVTree.computeLinTerms

SOVTree.**computeLinTerms**(*net*, *sov_data=None*, *eps=0.0001*)

Construct linear terms for *net* so that transfer impedance to soma is exactly matched

Parameters

- **net** (*neat.NETree*) – the neural evaluation tree (NET)
- **sov_data** (*None or tuple of mode matrices*) – If sov_data is not None, it is a tuple of a vector of the reciprocals of the mode timescales and a matrix with the corresponding spatial mode functions.
- **eps** (*float*) – the cutoff threshold in relative importance below which modes are truncated

Returns **lin_terms** – the kernels associated with linear terms of the NET, keys are indices of their corresponding location stored inder 'net eval'

Return type dict of {int: *neat.Kernel*}

class SOVNode(*index*, *p3d=None*)

Node that defines functions and stores quantities to implement separation of variables calculation (Major, 1993)

Greens Tree

class GreensTree(*file_n=None*, *types=[1, 3, 4]*)

Class that computes the Green's function in the Fourrier domain of a given neuronal morphology (Koch, 1985). This three defines a special *neat.SomaGreensNode* as a derived class from *neat.GreensNode* as some functions required for Green's function calculation are different and thus overwritten.

The calculation proceeds on the computational tree (see docstring of *neat.MorphNode*). Thus it makes no sense to look for Green's function related quantities in the original tree.

Variables **freqs** (*np.array of complex*) – Frequencies at which impedances are evaluated [Hz]

GreensTree.removeExpansionPoints()	Remove expansion points from all nodes in the tree
GreensTree.setImpedance(*args, **kwargs)	Set the boundary impedances for each node in the tree
GreensTree.calcZF(*args, **kwargs)	Computes the transfer impedance between two locations for all frequencies in *self.freqs*.
GreensTree.calcImpedanceMatrix(*args, **kwargs)	Computes the impedance matrix of a given set of locations for each frequency stored in *self.freqs*.

neat.GreensTree.removeExpansionPoints

GreensTree.**removeExpansionPoints**()
Remove expansion points from all nodes in the tree

neat.GreensTree.setImpedance

GreensTree.**setImpedance**(**args*, ***kwargs*)
Set the boundary impedances for each node in the tree

Parameters

- **freqs** (*np.ndarray* (dtype=complex, ndim=1)) – frequencies at which the impedances will be evaluated [Hz]
- **use_conc** (*bool*) – whether or not to incorporate concentrations in the calculation
- **pprint** (bool (default False)) – whether or not to print info on the progression of the algorithm

neat.GreensTree.calcZF

GreensTree.**calcZF**(**args*, ***kwargs*)
Computes the transfer impedance between two locations for all frequencies in *self.freqs*.

Parameters

- **loc1** (dict, tuple or *:class:MorphLoc*) – One of two locations between which the transfer impedance is computed
- **loc2** (dict, tuple or *:class:MorphLoc*) – One of two locations between which the transfer impedance is computed

Returns The transfer impedance [MOhm] as a function of frequency

Return type nd.ndarray (dtype = complex, ndim = 1)

neat.GreensTree.calcImpedanceMatrix

GreensTree.**calcImpedanceMatrix**(**args*, ***kwargs*)
Computes the impedance matrix of a given set of locations for each frequency stored in *self.freqs*.

Parameters

- **locarg** (*list* of locations or string) – if *list* of locations, specifies the locations for which the impedance matrix is evaluated, if string, specifies the name under which a set of location is stored
- **explicit_method** (bool, optional (default True)) – if False, will use the transitivity property of the impedance matrix to further optimize the computation.

Returns the impedance matrix, first dimension corresponds to the frequency, second and third dimensions contain the impedance matrix [MOhm] at that frequency

Return type *np.ndarray* (dtype = self.freqs.dtype, ndim = 3)

class GreensNode(*index*, *p3d*)

Node that stores quantities and defines functions to implement the impedance matrix calculation based on Koch's algorithm (Koch & Poggio, 1985).

Variables **expansion_points** (*dict {str: np.ndarray}*) – Stores ion channel expansion points for this segment.

GreensNode.setExpansionPoint(channel_name, ...)	Set the choice for the state variables of the ion channel around which to linearize.

neat.GreensNode.setExpansionPoint

GreensNode.**setExpansionPoint**(*channel_name*, *statevar*)

Set the choice for the state variables of the ion channel around which to linearize.

Note that when adding an ion channel to the node, the default expansion point setting is to linearize around the asymptotic values for the state variables at the equilibrium potential store in *self.e_eq*. Hence, this function only needs to be called to change that setting.

Parameters

- **channel_name** (*string*) – the name of the ion channel
- **statevar** (*dict*) – The expansion points for each of the ion channel state variables

Simulate NEURON models

class NeuronSimTree(*file_n=None*, *types=[1, 3, 4]*, *factor_lambda=1.0*, *t_calibrate=0.0*, *dt=0.025*, *v_init=- 75.0*)

Tree class to define NEURON (Carnevale & Hines, 2004) based on *neat.PhysTree*.

Variables

- **sections** (*dict of hoc sections*) – Storage for hoc sections. Keys are node indices.
- **shunts** (*list of hoc mechanisms*) – Storage container for shunts
- **syns** (*list of hoc mechanisms*) – Storage container for synapses
- **iclamps** (*list of hoc mechanisms*) – Storage container for current clamps
- **vclamps** (*lis of hoc mechanisms*) – Storage container for voltage clamps
- **vecstims** (*list of hoc mechanisms*) – Storage container for vecstim objects
- **netcons** (*list of hoc mechanisms*) – Storage container for netcon objects
- **vecs** (*list of hoc vectors*) – Storage container for hoc spike vectors
- **dt** (*float*) – timestep of the simulator [ms]
- **t_calibrate** (*float*) – Time for the model to equilibrate``[ms]``. Not counted as part of the simulation.
- **factor_lambda** (*int or float*) – If int, the number of segments per section. If float, multiplies the number of segments given by the standard lambda rule (Carnevale, 2004) to give the number of compartments simulated (default value 1. gives the number given by the lambda rule)
- **v_init** (*float*) – The initial voltage at which the model is initialized [mV]

- **NeuronSimTree can be extended easily with custom point process mechanisms.** (*A*) –
- **make sure that you store the point process in an existing appropriate** (*Just*) –
- **container or in a custom storage container, since if all references** (*storage*) –
- **the hocobject disappear, the object itself will be deleted as well.** (*to*) –
- **code-block:** (**.**) – python: class CustomSimTree(NeuronSimTree): def addCustomPointProcessMech(self, loc, **kwargs):

  loc = MorphLoc(loc, self)

  # create the point process pp = h.custom_point_process(self.sections[loc['node']](loc['x'])) pp.arg1 = kwargs['arg1'] pp.arg2 = kwargs['arg2'] ...

  self.storage_container_for_point_process.append(pp)
- **you define a custom storage container, make sure that you overwrite the** (*If*) –
- **and deleteModel() functions to make sure it is created and** (*`__init__()`*) –
- **properly.** (*deleted*) –

NeuronSimTree.initModel([dt, t_calibrate, ...])	Initialize hoc-objects to simulate the neuron model implemented by this tree.
NeuronSimTree.deleteModel()	Delete all stored hoc-objects
NeuronSimTree.addShunt(loc, g, e_r)	Adds a static conductance at a given location
NeuronSimTree.addDoubleExpCurrent(loc, tau1, ...)	Adds a double exponential input current at a given location
NeuronSimTree.addExpSynapse(loc, tau, e_r)	Adds a single-exponential conductance-based synapse
NeuronSimTree.addDoubleExpSynapse(loc, tau1, ...)	Adds a double-exponential conductance-based synapse
NeuronSimTree.addNMDASynapse(loc, tau, tau_nmda)	Adds a single-exponential conductance-based synapse with an AMPA and an NMDA component
NeuronSimTree.addDoubleExpNMDASynapse(loc, ...)	Adds a double-exponential conductance-based synapse with an AMPA and an NMDA component
NeuronSimTree.addIClamp(loc, amp, delay, dur)	Injects a DC current step at a given lcoation
NeuronSimTree.addSinClamp(loc, amp, delay, ...)	Injects a sinusoidal current at a given lcoation
NeuronSimTree.addOUClamp(loc, tau, mean, ...)	Injects a Ornstein-Uhlenbeck current at a given lcoation
NeuronSimTree.addOUconductance(loc, tau, ...)	Injects a Ornstein-Uhlenbeck conductance at a given location
NeuronSimTree.addOUReversal(loc, tau, mean, ...)	
NeuronSimTree.addVClamp(loc, e_c, dur)	Adds a voltage clamp at a given location
NeuronSimTree.setSpikeTrain(syn_index, ...)	Each hoc point process that receive spikes through should by appended to the synapse stack (stored under the list *self.syns*).
NeuronSimTree.run(t_max[, downsample, ...])	Run the NEURON simulation.

continues on next page

Table 20 – continued from previous page

NeuronSimTree.calcEEq([t_dur, set_e_eq])	Compute the equilibrium potentials in the middle (x=0.5) of each node.

neat.NeuronSimTree.initModel

NeuronSimTree.**initModel**(*dt=0.025*, *t_calibrate=0.0*, *v_init=-75.0*, *factor_lambda=1.0*, *pprint=False*)

Initialize hoc-objects to simulate the neuron model implemented by this tree.

Parameters

- **dt** (float (default is .025 ms)) – Timestep of the simulation
- **t_calibrate** (float (default 0. ms)) – The calibration time; time model runs without input to reach its equilibrium state before the true simulation starts
- **v_init** (float (default -75. mV)) – The initial voltage at which the model is initialized
- **factor_lambda** (*float or int (default 1.)*) – If int, the number of segments per section. If float, multiplies the number of segments given by the standard lambda rule (Carnevale, 2004) to give the number of compartments simulated (default value 1. gives the number given by the lambda rule)
- **pprint** (bool (default False)) – Whether or not to print info on the NEURON model's creation

neat.NeuronSimTree.deleteModel

NeuronSimTree.**deleteModel**()

Delete all stored hoc-objects

neat.NeuronSimTree.addShunt

NeuronSimTree.**addShunt**(*loc*, *g*, *e_r*)

Adds a static conductance at a given location

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the shunt.
- **g** (*float*) – The conductance of the shunt (uS)
- **e_r** (*float*) – The reversal potential of the shunt (mV)

neat.NeuronSimTree.addDoubleExpCurrent

NeuronSimTree.**addDoubleExpCurrent**(*loc*, *tau1*, *tau2*)

Adds a double exponential input current at a given location

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the current.
- **tau1** (*float*) – Rise time of the current waveform (ms)
- **tau2** (*float*) – Decay time of the current waveform (ms)

neat.NeuronSimTree.addExpSynapse

NeuronSimTree.**addExpSynapse**(*loc*, *tau*, *e_r*)
Adds a single-exponential conductance-based synapse

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the current.
- **tau** (*float*) – Decay time of the conductance window (ms)
- **e_r** (*float*) – Reversal potential of the synapse (mV)

neat.NeuronSimTree.addDoubleExpSynapse

NeuronSimTree.**addDoubleExpSynapse**(*loc*, *tau1*, *tau2*, *e_r*)
Adds a double-exponential conductance-based synapse

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the current.
- **tau1** (*float*) – Rise time of the conductance window (ms)
- **tau2** (*float*) – Decay time of the conductance window (ms)
- **e_r** (*float*) – Reversal potential of the synapse (mV)

neat.NeuronSimTree.addNMDASynapse

NeuronSimTree.**addNMDASynapse**(*loc*, *tau*, *tau_nmda*, *e_r=0.0*, *nmda_ratio=1.7*)
Adds a single-exponential conductance-based synapse with an AMPA and an NMDA component

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the current.
- **tau** (*float*) – Decay time of the AMPA conductance window (ms)
- **tau_nmda** (*float*) – Decay time of the NMDA conductance window (ms)
- **e_r** (float (optional, default `0.` mV)) – Reversal potential of the synapse (mV)
- **nmda_ratio** (*float (optional, default 1.7)*) – The ratio of the NMDA over AMPA component. Means that the maximum of the NMDA conductance window is `nmda_ratio` times the maximum of the AMPA conductance window.

neat.NeuronSimTree.addDoubleExpNMDASynapse

NeuronSimTree.**addDoubleExpNMDASynapse**(*loc*, *tau1*, *tau2*, *tau1_nmda*, *tau2_nmda*, *e_r=0.0*, *nmda_ratio=1.7*)
Adds a double-exponential conductance-based synapse with an AMPA and an NMDA component

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the current.
- **tau1** (*float*) – Rise time of the AMPA conductance window (ms)
- **tau2** (*float*) – Decay time of the AMPA conductance window (ms)

- **tau1_nmda** (*float*) – Rise time of the NMDA conductance window (ms)
- **tau2_nmda** (*float*) – Decay time of the NMDA conductance window (ms)
- **e_r** (float (optional, default `0.` mV)) – Reversal potential of the synapse (mV)
- **nmda_ratio** (*float (optional, default 1.7)*) – The ratio of the NMDA over AMPA component. Means that the maximum of the NMDA conductance window is `nmda_ratio` times the maximum of the AMPA conductance window.

neat.NeuronSimTree.addIClamp

`NeuronSimTree.`**`addIClamp`**(*loc*, *amp*, *delay*, *dur*)

Injects a DC current step at a given lcoation

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the current.
- **amp** (*float*) – The amplitude of the current (nA)
- **delay** (*float*) – The delay of the current step onset (ms)
- **dur** (*float*) – The duration of the current step (ms)

neat.NeuronSimTree.addSinClamp

`NeuronSimTree.`**`addSinClamp`**(*loc*, *amp*, *delay*, *dur*, *bias*, *freq*, *phase*)

Injects a sinusoidal current at a given lcoation

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the current.
- **amp** (*float*) – The amplitude of the current (nA)
- **delay** (*float*) – The delay of the current onset (ms)
- **dur** (*float*) – The duration of the current (ms)
- **bias** (*float*) – Constant baseline added to the sinusoidal waveform (nA)
- **freq** (*float*) – Frequency of the sinusoid (Hz)
- **phase** (*float*) – Phase of the sinusoid (rad)

neat.NeuronSimTree.addOUClamp

`NeuronSimTree.`**`addOUClamp`**(*loc*, *tau*, *mean*, *stdev*, *delay*, *dur*, *seed=None*)

Injects a Ornstein-Uhlenbeck current at a given lcoation

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the current.
- **tau** (*float*) – Time-scale of the OU process (ms)
- **mean** (*float*) – Mean of the OU process (nA)
- **stdev** (*float*) – Standard deviation of the OU process (nA)
- **delay** (*float*) – The delay of current onset from the start of the simulation (ms)

- **dur** (*float*) – The duration of the current input (ms)
- **seed** (*int, optional*) – Seed for the random number generator

neat.NeuronSimTree.addOUconductance

NeuronSimTree.**addOUconductance**(*loc*, *tau*, *mean*, *stdev*, *e_r*, *delay*, *dur*, *seed=None*)

Injects a Ornstein-Uhlenbeck conductance at a given location

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the conductance.
- **tau** (*float*) – Time-scale of the OU process (ms)
- **mean** (*float*) – Mean of the OU process (uS)
- **stdev** (*float*) – Standard deviation of the OU process (uS)
- **e_r** (*float*) – Reversal of the current (mV)
- **delay** (*float*) – The delay of current onset from the start of the simulation (ms)
- **dur** (*float*) – The duration of the current input (ms)
- **seed** (*int, optional*) – Seed for the random number generator

neat.NeuronSimTree.addOUReversal

NeuronSimTree.**addOUReversal**(*loc*, *tau*, *mean*, *stdev*, *g_val*, *delay*, *dur*, *seed=None*)

neat.NeuronSimTree.addVClamp

NeuronSimTree.**addVClamp**(*loc*, *e_c*, *dur*)

Adds a voltage clamp at a given location

Parameters

- **loc** (dict, tuple or *neat.MorphLoc*) – The location of the conductance.
- **e_c** (*float*) – The clamping voltage (mV)
- **dur** (*float, ms*) – The duration of the voltage clamp

neat.NeuronSimTree.setSpikeTrain

NeuronSimTree.**setSpikeTrain**(*syn_index*, *syn_weight*, *spike_times*)

Each hoc point process that receive spikes through should by appended to the synapse stack (stored under the list *self.syns*).

Default NeuronSimTree point processes that are added to *self.syns* are: - *self.addDoubleExpCurrent()* - *self.addExpSyn()* - *self.addDoubleExpSyn()* - *self.addDoubleExpSyn()* - *self.addNMDASynapse()* - *self.addDoubleExpNMDASynapse()*

With this function, these synapse can be set to receive a specific spike train.

Parameters

- **syn_index** (*int*) – index of the point process in the synapse stack

- **syn_weight** (*float*) – weight of the synapse (maximal value of the conductance window)
- **spike_times** (list or *np.array* of floats) – the spike times

neat.NeuronSimTree.run

NeuronSimTree.**run**(*t_max*, *downsample=1*, *record_from_syns=False*, *record_from_iclamps=False*, *record_from_vclamps=False*, *record_from_channels=False*, *record_v_deriv=False*, *record_concentrations=[]*, *pprint=False*)

Run the NEURON simulation. Records at all locations stored under the name 'rec locs' on *self* (see *MorphTree.storeLocs()*)

Parameters

- **t_max** (*float*) – Duration of the simulation
- **downsample** (*int (> 0)*) – Records the state of the model every *downsample* timesteps
- **record_from_syns** (bool (default `False`)) – Record currents of synapstic point processes (in *self.syns*). Accessible as *np.ndarray* in the output dict under key 'i_syn'
- **record_from_iclamps** (bool (default `False`)) – Record currents of iclamps (in *self.iclamps*) Accessible as *np.ndarray* in the output dict under key 'i_clamp'
- **record_from_vclamps** (bool (default `False`)) – Record currents of vclamps (in *self.vclamps*) Accessible as *np.ndarray* in the output dict under key 'i_vclamp'
- **record_from_channels** (bool (default `False`)) – Record channel state variables from *neat* defined channels in *self*, at locations stored under 'rec locs' Accessible as *np.ndarray* in the output dict under key 'chan'
- **record_v_deriv** (bool (default `False`)) – Record voltage derivative at locations stored under 'rec locs' Accessible as *np.ndarray* in the output dict under key 'dv_dt'
- **record_from_concentrations** (bool (default `False`)) – Record ion concentration at locations stored under 'rec locs' Accessible as *np.ndarray* in the output dict with as key the ion's name

Returns Dictionary with the results of the simulation. Contains time and voltage as *np.ndarray* at locations stored under the name ' rec locs', respectively with keys 't' and 'v_m'. Also contains traces of other recorded variables if the option to record them was set to `True`

Return type dict

neat.NeuronSimTree.calcEEq

NeuronSimTree.**calcEEq**(*t_dur=100.0*, *set_e_eq=True*)

Compute the equilibrium potentials in the middle (`x=0.5`) of each node.

Parameters

- **t_dur** (float (optional, default `100.` ms)) – The duration of the simulation
- **set_e_eq** (bool (optional, default `True`)) – Store the equilibrium potential as the `PhysNode.e_eq` attribute

6.3.3 Other Classes

Fitting reduced models

class CompartmentFitter(*phys_tree*, *e_hs=array([- 75.0, - 55.0, - 35.0, - 15.0])*, *freqs=array([0.0])*, *name='dont save'*, *path=''*)

Helper class to streamline fitting reduced compartmental models

Variables

- **tree** (*neat.PhysTree()*) – The full tree based on which reductions are made
- **e_hs** (*np.array of float*) – The holding potentials for which quasi active expansions are computed
- **freqs** (np.array of float or complex (default is `np.array([0.])`)) – The frequencies at which impedance matrices are evaluated
- **name** (*str (default 'dont save')*) – name of files in which intermediate trees required for the fit are stored. Details about what is in the actual pickle files are appended as a suffix to *name*. Default is to not store intermediate files.
- **path** (*str (default '')*) – specify a path under which the intermediate files are saved (only if *name* is specified). Default is empty string, which means that intermediate files are stored in the working directory.

To implement the default methodology.

CompartmentFitter.setCTree(loc_arg[, ...])	Store an initial *neat.CompartmentTree*, providing a tree structure scaffold for the fit for a given set of locations.
CompartmentFitter.fitModel(loc_arg[, ...])	Runs the full fit for a set of locations (the location are automatically extended with the bifurcation locs)

neat.CompartmentFitter.setCTree

CompartmentFitter.**setCTree**(*loc_arg*, *extend_w_bifurc=True*)

Store an initial *neat.CompartmentTree*, providing a tree structure scaffold for the fit for a given set of locations. The locations are also stored on `self.tree` under the name 'fit locs'

Parameters

- **loc_arg** (*list of locations or string (see documentation of MorphTree._convertLocArgToLocs()* for details) The compartment locations
- **extend_w_bifurc** (bool (optional, default *True*)) – To extend the compartment locations with all intermediate bifurcations (see documentation of *MorphTree.extendWithBifurcationLocs()*).

neat.CompartmentFitter.fitModel

CompartmentFitter.**fitModel**(*loc_arg*, *alpha_inds=[0]*, *use_all_channels_for_passive=True*, *recompute=False*, *pprint=False*, *parallel=False*)

Runs the full fit for a set of locations (the location are automatically extended with the bifurcation locs)

Parameters

- **loc_arg** (*list of locations or string (see documentation of* – *MorphTree._convertLocArgToLocs()* for details) The compartment locations
- **alpha_inds** (*list of ints*) – Indices of all mode time-scales to be included in the fit
- **use_all_channels_for_passive** (bool (optional, default True)) – Uses all channels in the tree to compute coupling conductances
- **recompute** (*bool*) – whether to force recomputing the impedances
- **pprint** (*bool*) – whether to print information
- **parallel** (*bool*) – whether the models are evaluated in parallel

Returns The reduced tree containing the fitted parameters

Return type *neat.CompartmentTree*

To check the faithfullness of the passive reduction, the following functions implement vizualisation of impedance kernels.

CompartmentFitter.checkPassive(loc_arg[, ...])	Checks the impedance kernels of the passive model.
CompartmentFitter.getKernels([recompute, pprint])	Returns the impedance kernels as a double nested list of "neat.Kernel".
CompartmentFitter.plotKernels([alphas, ...])	Plots the impedance kernels.

neat.CompartmentFitter.checkPassive

CompartmentFitter.**checkPassive**(*loc_arg*, *alpha_inds=[0]*, *n_modes=5*, *use_all_channels_for_passive=True*, *force_tau_m_fit=False*, *recompute=False*, *pprint=False*)

Checks the impedance kernels of the passive model.

Parameters

- **loc_arg** (*list of locations or string (see documentation of* – *MorphTree._convertLocArgToLocs()* for details) The compartment locations
- **alpha_inds** (*list of ints*) – Indices of all mode time-scales to be included in the fit
- **n_modes** (*int*) – The number of eigen modes that are shown
- **use_all_channels_for_passive** (*bool*) – Uses all channels in the tree to compute coupling conductances
- **force_tau_m_fit** (*bool*) – Force using the local membrane time-scale for capacitance fit
- **recompute** (*bool*) – whether to force recomputing the impedances

- **pprint** (*bool*) – is verbose if `True`

Returns

Return type `None`

neat.CompartmentFitter.getKernels

`CompartmentFitter.`**`getKernels`**(*recompute=False*, *pprint=False*)

Returns the impedance kernels as a double nested list of "neat.Kernel". The element at the position i,j represents the transfer impedance kernel between compartments i and j.

Parameters

- **recompute** (*bool*) – Force recomputing the SOV expansion if `True`
- **pprint** (*bool*) – Is verbose if `True`

Returns

- **k_orig** (list of list of *neat.Kernel*) – The kernels of the full model
- **k_comp** (list of list of *neat.Kernel*) – The kernels of the reduced model

neat.CompartmentFitter.plotKernels

`CompartmentFitter.`**`plotKernels`**(*alphas=None*, *phimat=None*, *t_arr=None*, *recompute=False*, *pprint=False*)

Plots the impedance kernels. The kernel at the position i,j represents the transfer impedance kernel between compartments i and j.

Parameters

- **alphas** (*np.array*) – The exponential coefficients, as follows from the SOV expansion
- **phimat** (*np.ndarray (dim=2)*) – The matrix to compute the exponential prefactors, as follows from the SOV expansion
- **t_arr** (*np.array*) – The time-points at which the to be plotted kernels are evaluated. Default is `np.linspace(0.,200.,int(1e3))`
- **recompute** (*bool*) – Force recomputing the SOV expansion if `True` (only if *alphas* or *phimat* are `None`)
- **pprint** (*bool*) – Is verbose if `True`

Returns

- **k_orig** (list of list of *neat.Kernel*) – The kernels of the full model
- **k_comp** (list of list of *neat.Kernel*) – The kernels of the reduced model

Individual fit functions.

CompartmentFitter.createTreeGF([channel_names])	Create a *FitTreeGF* copy of the old tree, but only with the channels in `channel_names`.
CompartmentFitter.createTreeSOV([eps])	Create a *SOVTree* copy of the old tree
CompartmentFitter.fitPassiveLeak([...])	Fit leak only.

continues on next page

Table 23 – continued from previous page

CompartmentFitter.fitPassive([...])	Fit the steady state passive model, consisting only of leak and coupling conductances, but ensure that the coupling conductances takes the passive opening of all channels into account
CompartmentFitter.evalChannel(channel_name)	Evaluate the impedance matrix for the model restricted to a single ion channel type.
CompartmentFitter.fitChannels([recompute, ...])	Fit the active ion channel parameters
CompartmentFitter.fitCapacitance([inds, ...])	Fit the capacitances of the model to the largest SOV time scale
CompartmentFitter.setEEq([t_max, dt, ...])	Set equilibrium potentials, measured from neuron simulation.
CompartmentFitter.getEEq(e_eqs_type, **kwargs)	Get equilibrium potentials.
CompartmentFitter.fitEEq(**kwargs)	Fits the leak potentials of the reduced model to yield the same equilibrium potentials as the full model
CompartmentFitter.fitSynRescale(c_locarg, ...)	Computes the rescaled conductances when synapses are moved to compartment locations, assuming a given average conductance for each synapse.

neat.CompartmentFitter.createTreeGF

CompartmentFitter.**createTreeGF**(*channel_names=[]*)

Create a *FitTreeGF* copy of the old tree, but only with the channels in `channel_names`. Leak 'L' is included in the tree by default.

Parameters **channel_names** (*list of strings*) – List of channel names of the channels that are to be included in the new tree.

Returns

Return type *FitTreeGF()*

neat.CompartmentFitter.createTreeSOV

CompartmentFitter.**createTreeSOV**(*eps=1.0*)

Create a *SOVTree* copy of the old tree

Parameters **channel_names** (*list of strings*) – List of channel names of the channels that are to be included in the new tree

Returns

Return type *neat.tools.fittools.compartmentfitter.FitTreeSOV*

neat.CompartmentFitter.fitPassiveLeak

CompartmentFitter.**fitPassiveLeak**(*recompute=False*, *pprint=True*)

Fit leak only. Coupling conductances have to have been fit already.

Parameters

- **recompute** (bool (optional, defaults to False)) – whether to force recomputing the impedances
- **pprint** (bool (optional, defaults to False)) – whether to print information

neat.CompartmentFitter.fitPassive

CompartmentFitter.**fitPassive**(*use_all_channels=True*, *recompute=False*, *pprint=False*)

Fit the steady state passive model, consisting only of leak and coupling conductances, but ensure that the coupling conductances takes the passive opening of all channels into account

Parameters

- **use_all_channels** (*bool (optional)*) – use leak at rest of all channels combined in the passive fit (passive leak has to be refit after capacitance fit)
- **recompute** (bool (optional, defaults to False)) – whether to force recomputing the impedances
- **pprint** (bool (optional, defaults to False)) – whether to print information

neat.CompartmentFitter.evalChannel

CompartmentFitter.**evalChannel**(*channel_name*, *recompute=False*, *pprint=False*, *parallel=True*, *max_workers=None*)

Evaluate the impedance matrix for the model restricted to a single ion channel type.

Parameters

- **channel_name** (*string*) – The name of the ion channel under consideration
- **recompute** (bool (optional, defaults to False)) – whether to force recomputing the impedances
- **pprint** (bool (optional, defaults to False)) – whether to print information
- **parallel** (bool (optional, defaults to True)) – whether the models are evaluated in parallel

Returns

Return type fit_mats

neat.CompartmentFitter.fitChannels

CompartmentFitter.**fitChannels**(*recompute=False*, *pprint=False*, *parallel=True*)

Fit the active ion channel parameters

Parameters

- **recompute** (bool (optional, defaults to `False`)) – whether to force recomputing the impedances
- **pprint** (bool (optional, defaults to `False`)) – whether to print information
- **parallel** (bool (optional, defaults to `True`)) – whether the models are evaluated in parallel

neat.CompartmentFitter.fitCapacitance

CompartmentFitter.**fitCapacitance**(*inds=[0]*, *check_fit=True*, *force_tau_m_fit=False*, *recompute=False*, *pprint=False*, *pplot=False*)

Fit the capacitances of the model to the largest SOV time scale

Parameters

- **inds** (list of int (optional, defaults to `[0]`)) – indices of eigenmodes used in the fit. Default is [0], indicating the largest eigenmode
- **check_fit** (bool (optional, default `True`)) – Check whether the largest eigenmode of the reduced model is within tolerance of the largest eigenmode of the full tree. If not, capacitances are set to mach membrane time scale
- **force_tau_m_fit** (bool (optional, default `False`)) – force capacitance fit through membrance time scale matching
- **recompute** (bool (optional, defaults to `False`)) – whether to force recomputing the impedances
- **pprint** (bool (optional, defaults to `False`)) – whether to print information
- **pplot** (bool (optional, defaults to `False`)) – whether to plot the eigenmode timescales

neat.CompartmentFitter.setEEq

CompartmentFitter.**setEEq**(*t_max=500.0*, *dt=0.1*, *factor_lambda=10.0*)

Set equilibrium potentials, measured from neuron simulation. Sets the *v_eqs_tree* and *v_eqs_fit* attributes, respectively containing the equilibrium potentials at (the middle of) each node in the original tree and at each of the fit locations

Parameters

- **t_max** (*float*) – duration of the neuron simulation
- **dt** (*float*) – time-step of the neuron simulation
- **factor_lambda** (*int of float*) – if int, signifies the number of segments per section. If float, multiplies the number of segments given by the lambda rule with this number

neat.CompartmentFitter.getEEq

CompartmentFitter.**getEEq**(*e_eqs_type*, ***kwargs*)

Get equilibrium potentials. Specify *v_eqs_tree* and *v_eqs_fit* attributes, respectively containing the equilibrium potentials at (the middle of) each node in the original tree and at each of the fit locations

Parameters

- **e_eqs_type** (*'tree' or 'fit'*) – For 'tree', returns the *v_eqs_tree* attribute, containing the equilibrium potentials at (the middle of) each node in the original tree. For 'fit', returns the *v_eqs_fit* attribute, containing the equilibrium potentials at each of the fit locations.
- **kwargs** (When *v_eqs_tree* or *v_eqs_fit*, have not been set, calls) – ::func::*self.setEEq()* with these *kwargs*

neat.CompartmentFitter.fitEEq

CompartmentFitter.**fitEEq**(***kwargs*)

Fits the leak potentials of the reduced model to yield the same equilibrium potentials as the full model

Parameters **kwargs** (When *v_eqs_tree* or *v_eqs_fit*, have not been set, calls) – ::func::*self.setEEq()* with these *kwargs*

neat.CompartmentFitter.fitSynRescale

CompartmentFitter.**fitSynRescale**(*c_locarg*, *s_locarg*, *comp_inds*, *g_syns*, *e_revs*, *fit_impedance=False*, *channel_names=None*, *recompute=False*)

Computes the rescaled conductances when synapses are moved to compartment locations, assuming a given average conductance for each synapse.

Parameters

- **c_locarg** (*list of locations or string (see documentation of MorphTree._convertLocArgToLocs()* for details) The compartment locations
- **s_locarg** (*list of locations or string (see documentation of MorphTree._convertLocArgToLocs()* for details) The synapse locations
- **comp_inds** (*list or numpy.array of ints*) – for each location in [s_locarg], gives the index of the compartment location in [c_locarg] to which the synapse is assigned
- **g_syns** (*list or numpy.array of floats*) – The average conductances for each synapse
- **e_revs** (*list or numpy.array of floats*) – The reversal potential of each synapse
- **fit_impdedance** (bool (optional, default *False*)) – Whether to also use the reproduction of the rescaled impedance matrix as target.
- **channel_names** (list of str or *None* (default)) – List of ion channels to be included in impedance matrix calculation. *None* includes all ion channels
- **recompute** (bool (defaults is *False*)) – Whether or not to recompute the impedance tree for this channel configuration

Returns **g_resc** – The rescale values for the synaptic weights

Return type numpy.array of floats

Defining ion channels

class IonChannel

Base ion channel class that implements linearization and code generation for NEURON (.mod-files) and C++.

Userdefined ion channels should inherit from this class and implement the *define()* function, where the specific attributes of the ion channel are set.

The ion channel current is of the form

$$i_{chan} = \overline{g}\, p_o(x_1, ..., x_n)\, (e - v)$$

where p_o is the open probability defined as a function of a number of state variables. State variables evolve according to

$$\dot{x}_i = f_i(x_i, v, c_1, ..., c_k)$$

with $c_1, ..., c_n$ the (optional) set of concentrations the ion channel depends on. There are two canonical ways to define f_i, either based on reaction rates α and β:

$$\dot{x}_i = \alpha_i(v)\,(1 - x_i) - \beta_i(v)\,x_i,$$

or based on an asymptotic value x_i^∞ and time-scale τ_i

$$\dot{x}_i = \frac{x_i^\infty(v) - x_i}{\tau_i(v)}.$$

IonChannel accepts handles either description. For the former description, dicts *self.alpha* and *self.beta* must be defined with as keys the names of every state variable in the open probability. Similarly, for the latter description, dicts *self.tauinf* and *self.varinf* must be defined with as keys the name of every state variable.

The user **must** define the attributes *p_open*, and either *alpha* and *beta* or *tauinf* and *varinf* in the *define()* function. The other attributes *ion*, *conc*, *q10*, *temp*, and *e* are optional.

Parameters

- **p_open** (*str*) – The open probability of the ion channel.
- **alpha** (*dict {str: str}*) – dictionary of the rate function for each state variables. Keys must correspond to the name of every state variable in *p_open*, values must be formulas written as strings with *v* and possible ion as variabels
- **beta** (*dict {str: str}*) – dictionary of the rate function for each state variables. Keys must correspond to the name of every state variable in *p_open*, values must be formulas written as strings with *v* and possible ion as variabels
- **tauinf** (*dict {str: str}*) – state variable time scale and asymptotic activation level. Keys must correspond to the name of every state variable in *p_open*, values must be formulas written as strings with *v* and possible ion as variabels
- **varinf** (*dict {str: str}*) – state variable time scale and asymptotic activation level. Keys must correspond to the name of every state variable in *p_open*, values must be formulas written as strings with *v* and possible ion as variabels
- **ion** (*str ('na', 'ca', 'k' or ''), optional*) – The ion to which the ion channel is permeable

- **conc** (*set of str (containing 'na', 'ca', 'k') or dict of {str: float}*) – The concentrations the ion channel activation depends on. Can be a set of ions or a dict with the ions as keys and default values as float.
- **q10** (*str, optional*) – Temperature dependence of the state variable rate functions. May be a float or a string convertible to a sympy expression containing the *temp* parameter (temperature in `[deg C]`). This factor divides the time-scales :math:` au_i(v)` of the ion channel. If not given, default is 1.
- **temp** (*float, optional*) – The temperature at which the ion channel is evaluated. Can be modified after initializiation by calling *IonChannel.setDefaultParams(temp=new_temperature)*. If not given, the evaluates *self.q10* at the default temperature of 36 degC.
- **e** (*float, optional*) – Reversal of the ion channel in `[mV]`. functions that need it allow the default value to be overwritten with a keyword argument. If nothing is provided, will take a default reversal for *self.ion* (which is -85 mV for 'K', 50 mV for 'Na' and 50 mV for 'Ca'). If no ion is provided, errors will occur if functions that need *e* are called without specifying the value as a keyword argument.

Examples

```
>>> class Na_Ta(IonChannel):
>>>     def define(self):
>>>         # from (Colbert and Pan, 2002), Used in (Hay, 2011)
>>>         self.ion = 'na'
>>>         # concentrations the ion channel depends on
>>>         self.conc = {}
>>>         # define channel open probability
>>>         self.p_open = 'h * m ** 3'
>>>         # define activation functions
>>>         self.alpha, self.beta = {}, {}
>>>         self.alpha['m'] =  '0.182 * (v + 38.) / (1. - exp(-(v + 38.) / 6.))'
 ↪# 1/ms
>>>         self.beta['m']  = '-0.124 * (v + 38.) / (1. - exp( (v + 38.) / 6.))'
 ↪# 1/ms
>>>         self.alpha['h'] = '-0.015 * (v + 66.) / (1. - exp( (v + 66.) / 6.))'
 ↪# 1/ms
>>>         self.beta['h']  =  '0.015 * (v + 66.) / (1. - exp(-(v + 66.) / 6.))'
 ↪# 1/ms
>>>         # temperature factor for reaction rates
>>>         self.q10 = '2.3^((temp - 23.)/10.)'
```

IonChannel.setDefaultParams(**kwargs)	**kwargs
IonChannel.computePOpen(v, **kwargs)	Compute the open probability of the ion channel
IonChannel.computeDerivatives(v, **kwargs)	Compute: (i) the derivatives of the open probability to the state variables (ii) The derivatives of state functions to the voltage (iii) The derivatives of state functions to the state variables
IonChannel.computeDerivativesConc(v, **kwargs)	Compute the derivatives of the state functions to the concentrations
IonChannel.computeVarinf(v)	Compute the asymptotic values for the state variables at a given activation level

continues on next page

Table 24 – continued from previous page

IonChannel.computeTauinf(v)	Compute the time-scales for the state variables at a given activation level
IonChannel.computeLinear(v, freqs, **kwargs)	Combute the contributions of the state variables to the linearized channel current
IonChannel.computeLinearConc(v, freqs, ion, ...)	Combute the contributions of the state variables to the linearized channel current
IonChannel.computeLinSum(v, freqs[, e])	Combute the linearized channel current contribution (without concentributions from the concentration - see *computeLinConc()*)
IonChannel.computeLinConc(v, freqs, ion[, e])	Combute the linearized channel current contribution from the concentrations

neat.IonChannel.setDefaultParams

IonChannel.**setDefaultParams**(***kwargs*)

****kwargs** Default values for temperature (*temp*), reversal (*e*)

neat.IonChannel.computePOpen

IonChannel.**computePOpen**(*v*, ***kwargs*)

Compute the open probability of the ion channel

Parameters

- **v** (float or *np.ndarray* of float) – The voltage at which to evaluate the open probability
- ****kwargs** – Optional values for the state variables and concentrations. Broadcastable to *v* if provided

Returns The open probability

Return type float or *np.ndarray* of float

neat.IonChannel.computeDerivatives

IonChannel.**computeDerivatives**(*v*, ***kwargs*)

Compute: (i) the derivatives of the open probability to the state variables (ii) The derivatives of state functions to the voltage (iii) The derivatives of state functions to the state variables

Parameters

- **v** (float or *np.ndarray*) – The voltage at which to evaluate the open probability
- ****kwargs** – Optional values for the state variables and concentrations. Broadcastable to *v* if provided

Returns The derivatives

Return type tuple of three floats or three `np.ndarray`s of float

neat.IonChannel.computeDerivativesConc

IonChannel.**computeDerivativesConc**(*v*, ***kwargs*)

Compute the derivatives of the state functions to the concentrations

Parameters

- **v** (float or *np.ndarray*) – The voltage at which to evaluate the open probability
- ****kwargs** – Optional values for the state variables and concentrations. Broadcastable to *v* if provided

Returns The derivatives

Return type tuple of three floats or three `np.ndarray`s of float

neat.IonChannel.computeVarinf

IonChannel.**computeVarinf**(*v*)

Compute the asymptotic values for the state variables at a given activation level

Parameters **v** (float or *np.ndarray*) – The voltage at which to evaluate the open probability

Returns The asymptotic activations, items are of same type (and shape) as *v*

Return type dict of *np.ndarray* of dict of float

neat.IonChannel.computeTauinf

IonChannel.**computeTauinf**(*v*)

Compute the time-scales for the state variables at a given activation level

Parameters **v** (float or *np.ndarray*) – The voltage at which to evaluate the open probability

Returns The asymptotic activations, items are of same type (and shape) as *v*

Return type dict of *np.ndarray* of dict of float

neat.IonChannel.computeLinear

IonChannel.**computeLinear**(*v*, *freqs*, ***kwargs*)

Combute the contributions of the state variables to the linearized channel current

Parameters

- **v** (float or *np.ndarray*) – The voltage `[mV]` at which to evaluate the open probability
- **float** (*freqs*) – The frequencies `[Hz]` at which to evaluate the linearized contribution
- **complex** – The frequencies `[Hz]` at which to evaluate the linearized contribution
- **np.ndarray of float or complex** (*or*) – The frequencies `[Hz]` at which to evaluate the linearized contribution
- ****kwargs** – Optional values for the state variables and concentrations. Broadcastable to *v* if provided

Returns The linearized current. Shape is dimension of *freqs* followed by the dimensions of *v*.

Return type float, complex or *np.ndarray* of float or complex

neat.IonChannel.computeLinearConc

IonChannel.**computeLinearConc**(*v*, *freqs*, *ion*, ***kwargs*)

Combute the contributions of the state variables to the linearized channel current

Parameters

- **v** (float or *np.ndarray*) – The voltage [mV] at which to evaluate the open probability
- **freqs** (float, complex, or *np.ndarray* of float or complex:) – The frequencies [Hz] at which to evaluate the linearized contribution
- **ion** (*str*) – The ion name for which to compute the linearized contribution
- ****kwargs** – Optional values for the state variables and concentrations. Broadcastable to *v* if provided

Returns The linearized current. Shape is dimension of *freqs* followed by the dimensions of *v*.

Return type float, complex or *np.ndarray* of float or complex

neat.IonChannel.computeLinSum

IonChannel.**computeLinSum**(*v*, *freqs*, *e=None*, ***kwargs*)

Combute the linearized channel current contribution (without concentributions from the concentration - see *computeLinConc()*)

Parameters

- **v** (float or *np.ndarray*) – The voltage [mV] at which to evaluate the open probability
- **freqs** (float, complex, or *np.ndarray* of float or complex:) – The frequencies [Hz] at which to evaluate the linearized contribution
- **e** (float or *None*) – The reversal potential of the channel. Defaults to the value stored in *self.default_params['e']* if not provided.
- ****kwargs** – Optional values for the state variables and concentrations. Broadcastable to *v* if provided

Returns The linearized current. Shape is dimension of *freqs* followed by the dimensions of *v*.

Return type float, complex or *np.ndarray* of float or complex

neat.IonChannel.computeLinConc

IonChannel.**computeLinConc**(*v*, *freqs*, *ion*, *e=None*, ***kwargs*)

Combute the linearized channel current contribution from the concentrations

Parameters

- **v** (float or *np.ndarray*) – The voltage [mV] at which to evaluate the open probability
- **freqs** (float, complex, or *np.ndarray* of float or complex:) – The frequencies [Hz] at which to evaluate the linearized contribution
- **ion** (*str*) – The ion name for which to compute the linearized contribution
- **e** (float or *None*) – The reversal potential of the channel. Defaults to the value stored in *self.default_params['e']* if not provided.

- ****kwargs** – Optional values for the state variables and concentrations. Broadcastable to *v* if provided

Returns The linearized current. Shape is dimension of *freqs* followed by the dimensions of *v*.

Return type float, complex or *np.ndarray* of float or complex

Neural evaluation tree simulator

Compute Fourrier transforms

class FourrierTools(*tarr*, *fmax=7*, *base=10*, *num=200*)

Performs an accurate Fourrier transform on functions evaluated at a given array of temporal grid points

Parameters

- **tarr** (*np.array* of floats,) – the time points (ms) at which the function is evaluated, have to be regularly spaced
- **fmax** (float, optional (default `7.`)) – the maximum value to which the logarithm is evaluated to get the maximum evaluation frequency
- **base** (float, optional (defaul `10`)) – the base of the logarithm used to generated the logspace
- **num** (int, even, optional (default `200`)) – Number of points. the eventual number of points in frequency space is (2+1/2)*num

Variables

- **s** (*np.array of complex*) – The frequencies at which input arrays in the Fourrier domain are supposed to be evaluated
- **t** (*np.array of real*) – The time array at which input arrays in the time domain are supposed to be evaluated
- **ind_0s** (*int*) – Index of the zero frequency component in *self.s*

FourrierTools.__call__(arr)	Evaluate the Fourrier transform of *arr*
FourrierTools.ft(arr)	Evaluate the Fourrier transform of *arr*
FourrierTools.ftInv(arr)	Evaluate the inverse Fourrier transform of *arr*

neat.FourrierTools.__call__

FourrierTools.**__call__**(*arr*)

Evaluate the Fourrier transform of *arr*

Parameters **arr** (*np.array*) – Should have the same length as *self.t*

Returns

- **s** (*np.array*) – the frequency points at which the Fourrier transform is evaluated (in Hz)
- **farr** (*np.array*) – the Fourrier transform of *arr*

neat.FourrierTools.ft

FourrierTools.**ft**(*arr*)
Evaluate the Fourrier transform of *arr*

Parameters **arr** (*np.array*) – Should have the same length as *self.t*

Returns

- **s** (*np.array*) – the frequency points at which the Fourrier transform is evaluated (in Hz)
- **farr** (*np.array*) – the Fourrier transform of *arr*

neat.FourrierTools.ftInv

FourrierTools.**ftInv**(*arr*)
Evaluate the inverse Fourrier transform of *arr*

Parameters **arr** (*np.array*) – Should have the same length as *self.s*

Returns

- **t** (*np.array*) – the time points at which the inverse Fourrier transform is evaluated (in ms)
- **tarr** (*np.array*) – the Fourrier transform of *arr*

6.4 Developer Guide

6.4.1 Developer overview

Willem Wybo Jakob Jordan Benjamin Ellenberger

6.4.2 Code of Conduct

Introduction

This code of conduct applies to all spaces managed by the NEAT project, including all public and private mailing lists, issue trackers, wikis, and any other communication channel used by our community.

This code of conduct should be honored by everyone who participates in the NEAT community formally or informally, or claims any affiliation with the project, in any project-related activities and especially when representing the project, in any role.

This code is not exhaustive or complete. It serves to distill our common understanding of a collaborative, shared environment and goals. Please try to follow this code in spirit as much as in letter, to create a friendly and productive environment that enriches the surrounding community.

Specific Guidelines

We strive to:

1. Be open. We invite anyone to participate in our community. We prefer to use public methods of communication for project-related messages, unless discussing something sensitive. This applies to messages for help or project-related support, too; not only is a public support request much more likely to result in an answer to a question, it also ensures that any inadvertent mistakes in answering are more easily detected and corrected.
2. Be empathetic, welcoming, friendly, and patient. We work together to resolve conflict, and assume good intentions. We may all experience some frustration from time to time, but we do not allow frustration to turn into a personal attack. A community where people feel uncomfortable or threatened is not a productive one.
3. Be collaborative. Our work will be used by other people, and in turn we will depend on the work of others. When we make something for the benefit of the project, we are willing to explain to others how it works, so that they can build on the work to make it even better. Any decision we make will affect users and colleagues, and we take those consequences seriously when making decisions.
4. Be inquisitive. Nobody knows everything! Asking questions early avoids many problems later, so we encourage questions, although we may direct them to the appropriate forum. We will try hard to be responsive and helpful.
5. Be careful in the words that we choose. We are careful and respectful in our communication and we take responsibility for our own speech. Be kind to others. Do not insult or put down other participants. We will not accept harassment or other exclusionary behaviour, such as:
    - Violent threats or language directed against another person.
    - Sexist, racist, or otherwise discriminatory jokes and language.
    - Posting sexually explicit or violent material.
    - Posting (or threatening to post) other people's personally identifying information ("doxing").
    - Sharing private content, such as emails sent privately or non-publicly, or unlogged forums such as IRC channel history, without the sender's consent.
    - Personal insults, especially those using racist or sexist terms.
    - Unwelcome sexual attention.
    - Excessive profanity. Please avoid swearwords; people differ greatly in their sensitivity to swearing.
    - Repeated harassment of others. In general, if someone asks you to stop, then stop.
    - Advocating for, or encouraging, any of the above behaviour.

Diversity Statement

The NEAT project welcomes and encourages participation by everyone. We are committed to being a community that everyone enjoys being part of. Although we may not always be able to accommodate each individual's preferences, we try our best to treat everyone kindly.

No matter how you identify yourself or how others perceive you: we welcome you. Though no list can hope to be comprehensive, we explicitly honour diversity in: age, culture, ethnicity, genotype, gender identity or expression, language, national origin, neurotype, phenotype, political beliefs, profession, race, religion, sexual orientation, socioeconomic status, subculture and technical ability.

Though we welcome people fluent in all languages, NEAT development is conducted in English.

Standards for behaviour in the NEAT community are detailed in the Code of Conduct above. Participants in our community should uphold these standards in all their interactions and help others to do so as well (see next section).

Reporting Guidelines

We know that it is painfully common for internet communication to start at or devolve into obvious and flagrant abuse. We also recognize that sometimes people may have a bad day, or be unaware of some of the guidelines in this Code of Conduct. Please keep this in mind when deciding on how to respond to a breach of this Code.

For clearly intentional breaches, report those to the Code of Conduct committee (see below). For possibly unintentional breaches, you may reply to the person and point out this code of conduct (either in public or in private, whatever is most appropriate). If you would prefer not to do that, please feel free to report to the Code of Conduct Committee directly, or ask the Committee for advice, in confidence.

You can report issues to the NEAT Code of Conduct committee.

If your report involves any members of the committee, or if they feel they have a conflict of interest in handling it, then they will recuse themselves from considering your report. Alternatively, if for any reason you feel uncomfortable making a report to the committee, then you can also contact:

- Senior NumFOCUS staff: conduct@numfocus.org.

Incident reporting resolution & Code of Conduct enforcement

We will investigate and respond to all complaints. The NEAT Code of Conduct Committee will protect the identity of the reporter, and treat the content of complaints as confidential (unless the reporter agrees otherwise).

In case of severe and obvious breaches, e.g., personal threat or violent, sexist or racist language, we will immediately disconnect the originator from NEAT communication channels.

In cases not involving clear severe and obvious breaches of this code of conduct, the process for acting on any received code of conduct violation report will be:

1. acknowledge report is received
2. reasonable discussion/feedback
3. mediation (if feedback didn't help, and only if both reporter and reportee agree to this)
4. enforcement via transparent decision by the Code of Conduct Committee

The committee will respond to any report as soon as possible, and at most within 72 hours.

Endnotes

This document is adapted from:

- SciPy Code of Conduct

6.4.3 Working with *NEAT* source code

Contents:

Introduction

These pages describe a git and github workflow for the neat project.

There are several different workflows here, for different ways of working with *neat*.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see *git resources*.

Install git

Overview

Debian / Ubuntu	`sudo apt-get install git`
Fedora	`sudo dnf install git`
Windows	Download and install msysGit
OS X	Use the git-osx-installer

In detail

See the git page for the most recent information.

Have a look at the github install help pages available from github help

There are good instructions here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Following the latest source

These are the instructions if you just want to follow the latest *neat* source, but you don't need to do any development for now.

The steps are:

- *Install git*
- get local copy of the neat github git repository
- update local copy from time to time

Get the local copy of the code

From the command line:

```
git clone git://github.com/neat/neat.git
```

You now have a copy of the code tree in the new `neat` directory.

Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd neat
git pull
```

The tree in `neat` will now have the latest changes from the initial repository.

Making a patch

You've discovered a bug or something else you want to change in neat .. — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the *Git for development* model instead.

Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/neat/neat.git
# make a branch for your patching
cd neat
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the neat mailing list — where we will thank you warmly.

In detail

1. Tell git who you are so it can label the commits you've made:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. If you don't already have one, clone a copy of the neat repository:

```
git clone git://github.com/neat/neat.git
cd neat
```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

```
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:

```
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

   Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag?.

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
git format-patch -M -C master
```

   You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

   Send these files to the neat mailing list.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the neat repository on github — *Making your own copy (fork) of neat*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/neat.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

Git for development

Contents:

Making your own copy (fork) of neat

You need to do this only once. The instructions here are very similar to the instructions at https://help.github.com/forking/ — please see that page for more detail. We're repeating some of it here just to give the specifics for the neat project, and to suggest some default names.

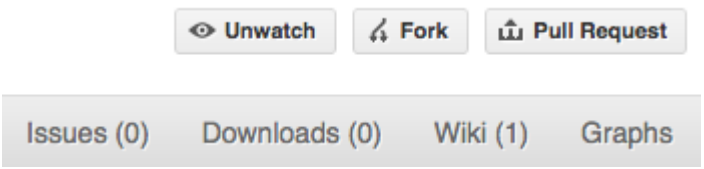
Set up and configure a github account

If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the `Generating SSH keys` help on github help.

Create your own forked copy of neat

1. Log into your github account.
2. Go to the neat github home at neat github.
3. Click on the *fork* button:



Now, after a short pause, you should find yourself at the home page for your own forked copy of neat.

Set up your fork

First you follow the instructions for *Making your own copy (fork) of neat*.

Overview

```
git clone git@github.com:your-user-name/neat.git
cd neat
git remote add upstream git://github.com/neat/neat.git
```

In detail

Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/neat.git`
2. Investigate. Change directory to your new repo: `cd neat`. Then `git branch -a` to show you all branches. You'll get something like:

   ```
   * master
   remotes/origin/master
   ```

   This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

   Now you want to connect to the upstream neat github repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

```
cd neat
git remote add upstream git://github.com/neat/neat.git
```

`upstream` here is just the arbitrary name we're using to refer to the main neat repository at neat github.

Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream     git://github.com/neat/neat.git (fetch)
upstream     git://github.com/neat/neat.git (push)
origin       git@github.com:your-user-name/neat.git (fetch)
origin       git@github.com:your-user-name/neat.git (push)
```

Configure git

Overview

Your personal git configurations are saved in the `.gitconfig` file in your home directory.

Here is an example `.gitconfig` file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com

[alias]
        ci = commit -a
        co = checkout
        st = status
        stat = status
        br = branch
        wdiff = diff --color-words

[core]
        editor = vim

[merge]
        summary = true
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

In detail

user.name and user.email

It is good practice to tell git who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
      name = Your Name
      email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words
```

Editor

You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging

To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

Fancy log output

This is a very nice alias to get a fancy log output; it should go in the `alias` section of your `.gitconfig` file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
↪%C(bold blue)[%an]%Creset' --abbrev-commit --date=relative
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45␣
↪minutes ago) [Matthew Brett]
*   d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/
↪master (2 weeks ago) [Jonathan Terhorst]
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2␣
↪weeks ago) [Corran Webster]
* 68f6752 - Initial implimentation of AxisIndexer - uses 'index_by' which needs to be␣
↪changed to a call on an Axes object - this is all very sketchy right now. (2 weeks␣
↪ago) [Corr
*   376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan␣
↪Terhorst]
|\
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-
↪axis object (3 weeks ago) [Jonathan Terhorst]
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan␣
↪Terhorst]
| *   956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago)␣
↪[Jonathan Terhorst]
| |\
| |/
```

Thanks to Yury V. Zaytsev for posting it.

Development workflow

You already have your own forked copy of the neat repository, by following *Making your own copy (fork) of neat*. You have *Set up your fork*. You have configured git by following *Configure git*. Now you are ready for some real work.

Workflow summary

In what follows we'll refer to the upstream neat `master` branch, as "trunk".

- Don't use your `master` branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — "one task, one branch" (ipython git workflow).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider *Rebasing on trunk*
- Ask on the neat mailing list if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

See linux git workflow and ipython git workflow for some explanation.

Consider deleting your master branch

It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See deleting master on github for details.

Update the mirror of trunk

First make sure you have done *Linking your repository to the upstream repo*.

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, 'trunk' is the branch referred to by (remote/branchname) `upstream/master` - and if there have been commits since you last checked, `upstream/master` will change after you do the fetch.

Make a new feature branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public github fork of neat. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

The editing workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
```

(continues on next page)

(continued from previous page)

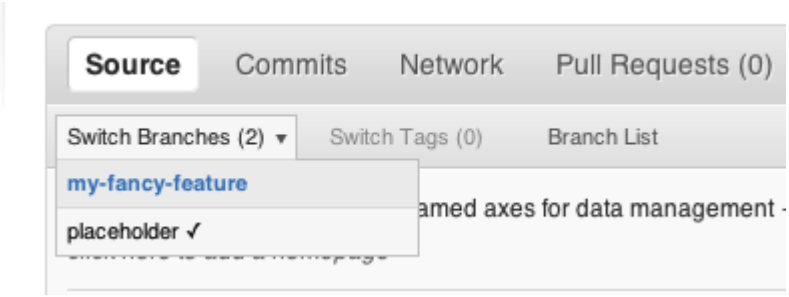
```
#  INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag? — and the helpful use-case description in the tangled working copy problem. The git commit manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see git push).

Ask for your changes to be reviewed or merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `https://github.com/your-user-name/neat`.
2. Use the 'Switch Branches' dropdown menu near the top left of the page to select the branch with your changes:



3. Click on the 'Pull request' button:

Enter a title for the set of changes, and some explanation of what you've done. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with.

If you don't think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

Some other things you might want to do

Delete a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

Note the colon `:` before `my-unwanted-branch`. See also: https://help.github.com/articles/pushing-to-a-remote/#deleting-a-remote-branch-or-tag

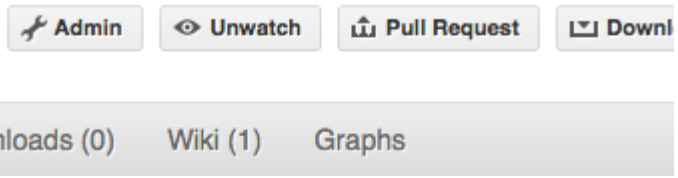
Several people sharing a single repository

If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork neat into your account, as from *Making your own copy (fork) of neat*.

Then, go to your forked repository github page, say `https://github.com/your-user-name/neat`

Click on the 'Admin' button, and add anyone else to the repo as a collaborator:



Now all those people can do:

```
git clone git@githhub.com:your-user-name/neat.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

Explore your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the network graph visualizer for your github repo.

Finally the *Fancy log output* `lg` alias will give you a reasonable text-based graph of the repository.

Rebasing on trunk

Let's say you thought of some work you'd like to do. You *Update the mirror of trunk* and *Make a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
     /
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
              A'--B'--C' cool-feature
             /
D---E---F---G trunk
```

See rebase without tears for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recovering from mess-ups*.

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the "Description" section. There is some related help on merging in the git user manual - see resolving a merge.

Recovering from mess-ups

Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto␣
↪11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...

# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

Note: Do this only for your own feature branches.

There's an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and `6ad92e5` is the last commit in the `cool-feature` branch. Suppose we want to make the following changes:

- Rewrite the commit message for `13d7934` to something more sensible.
- Combine the commits `2dec1ac`, `a815645`, `eadc391` into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for `13d7934`, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained *above*.

Maintainer workflow

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository.

Being as how you're a maintainer, you are completely on top of the basic stuff in *Development workflow*.

The instructions in *Linking your repository to the upstream repo* add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:neat/neat.git
git fetch upstream-rw
```

Integrating changes

Let's say you have some changes that need to go into trunk (`upstream-rw/master`).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/neat.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

A few commits

If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw
# rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

A long series of commits

If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history

Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

git resources

Tutorials and summaries

- github help has an excellent series of how-to guides.
- The pro git book is a good in-depth book on git.
- A git cheat sheet is a page giving summaries of common commands.
- The git user manual
- The git tutorial
- git ready — a nice series of tutorials
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- git foundation expands on the git parable.
- Fernando Perez' git page — Fernando's git page — many links and tips
- A good but technical page on git concepts
- git svn crash course: git for those of us used to subversion

Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management
- Linus Torvalds on linux git workflow . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add
- git branch
- git checkout
- git clone
- git commit
- git config
- git diff
- git log
- git pull
- git push
- git remote
- git status

6.5 Release Log

6.5.1 NEAT 0.9.1

Release date: 25 March 2021 Supports Python 3.5, 3.6, 3.7, and 3.8.

Release notes

See release/release_0.9.1.

6.5.2 NEAT 0.9.0

Release date: 20 March 2020 Supports Python 3.5, 3.6, 3.7, and 3.8.

Release notes

See release/release_0.9.0.

6.6 License

NEAT is distributed with the GNU General Public License.

```
            GNU GENERAL PUBLIC LICENSE

               Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                  Preamble

The GNU General Public License is a free, copyleft license for
software and other kinds of works.

The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you
these rights or asking you to surrender the rights.  Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether
gratis or for a fee, you must pass on to the recipients the same
freedoms that you received.  You must make sure that they, too, receive
or can get the source code.  And you must show them these terms so they
know their rights.

Developers that use the GNU GPL protect your rights with two steps:
(1) assert copyright on the software, and (2) offer you this License
```

(continues on next page)

(continued from previous page)

```
giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains
that there is no warranty for this free software.  For both users' and
authors' sake, the GPL requires that modified versions be marked as
changed, so that their problems will not be attributed erroneously to
authors of previous versions.

Some devices are designed to deny users access to install or run
modified versions of the software inside them, although the manufacturer
can do so.  This is fundamentally incompatible with the aim of
protecting users' freedom to change the software.  The systematic
pattern of such abuse occurs in the area of products for individuals to
use, which is precisely where it is most unacceptable.  Therefore, we
have designed this version of the GPL to prohibit the practice for those
products.  If such problems arise substantially in other domains, we
stand ready to extend this provision to those domains in future versions
of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents.
States should not allow patents to restrict development and use of
software on general-purpose computers, but in those that do, we wish to
avoid the special danger that patents applied to a free program could
make it effectively proprietary.  To prevent this, the GPL assures that
patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and
modification follow.

                TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this
License.  Each licensee is addressed as "you".  "Licensees" and
"recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy.  The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based
on the Program.

To "propagate" a work means to do anything with it that, without
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
computer or modifying a private copy.  Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.
```

(continues on next page)

(continued from previous page)

```
To "convey" a work means any kind of propagation that enables other
parties to make or receive copies.  Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License.  If
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
form of a work.

A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
subprograms and other parts of the work.

The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.

The Corresponding Source for a work in source code form is that
same work.

2. Basic Permissions.
```

(continues on next page)

(continued from previous page)

```
All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:
```

(continues on next page)

(continued from previous page)

```
 a) The work must carry prominent notices stating that you modified

it, and giving a relevant date.

 b) The work must carry prominent notices stating that it is

released under this License and any conditions added under section

7.  This requirement modifies the requirement in section 4 to

"keep intact all notices".

 c) You must license the entire work, as a whole, under this

License to anyone who comes into possession of a copy.  This

License will therefore apply, along with any applicable section 7

additional terms, to the whole of the work, and all its parts,

regardless of how they are packaged.  This License gives no

permission to license the work in any other way, but it does not

invalidate such permission if you have separately received it.

 d) If the work has interactive user interfaces, each must display

Appropriate Legal Notices; however, if the Program has interactive

interfaces that do not display Appropriate Legal Notices, your

work need not make them do so.

A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

 a) Convey the object code in, or embodied in, a physical product

(including a physical distribution medium), accompanied by the

Corresponding Source fixed on a durable physical medium
```

(continues on next page)

(continued from previous page)

```
customarily used for software interchange.

 b) Convey the object code in, or embodied in, a physical product
(including a physical distribution medium), accompanied by a
written offer, valid for at least three years and valid for as
long as you offer spare parts or customer support for that product
model, to give anyone who possesses the object code either (1) a
copy of the Corresponding Source for all the software in the
product that is covered by this License, on a durable physical
medium customarily used for software interchange, for a price no
more than your reasonable cost of physically performing this
conveying of source, or (2) access to copy the
Corresponding Source from a network server at no charge.

 c) Convey individual copies of the object code with a copy of the
written offer to provide the Corresponding Source.  This
alternative is allowed only occasionally and noncommercially, and
only if you received the object code with such an offer, in accord
with subsection 6b.

 d) Convey the object code by offering access from a designated
place (gratis or for a charge), and offer equivalent access to the
Corresponding Source in the same way through the same place at no
further charge.  You need not require recipients to copy the
Corresponding Source along with the object code.  If the place to
copy the object code is a network server, the Corresponding Source
may be on a different server (operated by you or a third party)
that supports equivalent copying facilities, provided you maintain
clear directions next to the object code saying where to find the
Corresponding Source.  Regardless of what server hosts the
Corresponding Source, you remain obligated to ensure that it is
```

(continues on next page)

(continued from previous page)

```
available for as long as needed to satisfy these requirements.

 e) Convey the object code using peer-to-peer transmission, provided

you inform other peers where the object code and Corresponding

Source of the work are being offered to the general public at no

charge under subsection 6d.

A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

"Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
```

(continues on next page)

(continued from previous page)

```
source code form), and must require no special password or key for
unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

 a) Disclaiming warranty or limiting liability differently from the

terms of sections 15 and 16 of this License; or

 b) Requiring preservation of specified reasonable legal notices or

author attributions in that material or in the Appropriate Legal

Notices displayed by works containing it; or

 c) Prohibiting misrepresentation of the origin of that material, or

requiring that modified versions of such material be marked in

reasonable ways as different from the original version; or

 d) Limiting the use for publicity purposes of names of licensors or

authors of the material; or

 e) Declining to grant rights under trademark law for use of some

trade names, trademarks, or service marks; or

 f) Requiring indemnification of licensors and authors of that

material by anyone who conveys the material (or modified versions of

it) with contractual assumptions of liability to the recipient, for

any liability that these contractual assumptions directly impose on
```

(continues on next page)

(continued from previous page)

```
those licensors and authors.

All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
```

(continues on next page)

(continued from previous page)

```
not accept this License.  Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.

If you convey a covered work, knowingly relying on a patent license,
```

(continues on next page)

(continued from previous page)

```
and the Corresponding Source of the work is not available for anyone
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
work and works based on it.

A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU Affero General Public License into a single
```

(continues on next page)

(continued from previous page)

```
combined work, and to convey the resulting work.  The terms of this
License will continue to apply to the part which is the covered work,
but the special requirements of the GNU Affero General Public License,
section 13, concerning interaction through a network will apply to the
combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of
the GNU General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU General Public License, you may choose any version ever published
by the Free Software Foundation.

If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.
```

(continues on next page)

(continued from previous page)

```
If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

               END OF TERMS AND CONDITIONS

       How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

 <one line to give the program's name and a brief idea of what it does.>

Copyright (C) <year>  <name of author>

 This program is free software: you can redistribute it and/or modify

it under the terms of the GNU General Public License as published by

the Free Software Foundation, either version 3 of the License, or

(at your option) any later version.

 This program is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the

GNU General Public License for more details.

 You should have received a copy of the GNU General Public License

along with this program.  If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

 <program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.

This is free software, and you are welcome to redistribute it

under certain conditions; type `show c' for details.
```

(continues on next page)

(continued from previous page)

```
The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<http://www.gnu.org/philosophy/why-not-lgpl.html>.
```

6.7 Credits

Willem Wybo

Jakob Jordan

Benjamin Ellenberger

Benjamin Torben-Nielsen

6.8 Citing

To cite NEAT please use the following publication:

Wybo, Willem A.M. et al. (2021) *Data-driven reduction of dendritic morphologies with preserved dendro-somatic responses*, eLife, 10:e60936, pp. 1–26, doi 10.7554/eLife.60936

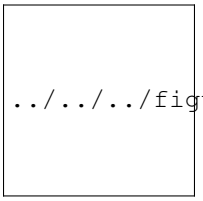
6.9 Bibliography

6.10 Examples

These are examples of how to obtain simplified compartmental models with NEAT. To run these examples, first compile the required ionchannels. From the *examples/* directory, run:

```
compilechannels models/channels/
```

6.10.1 Bac firing




```
import numpy as np
import warnings

from neat import MorphLoc, CompartmentFitter
from models.L5_pyramid import getL5Pyramid

from plotutil import *

import pickle

SIM_FLAG = 1
try:
    import neuron
    from neat import NeuronSimTree, NeuronCompartmentTree, createReducedNeuronModel
except ImportError:
    warnings.warn('NEURON not available, plotting stored image', UserWarning)
    SIM_FLAG = 0


## Parameters ######################################################################
# sites for simplification
D2S_CASPIKE = np.array([685., 785., 885., 985.])
D2S_APIC = np.array([85., 185., 285., 385., 485., 585.])
CA_LOC = (224, 0.86)

# morphology color map
vals = np.ones((2, 4))
vals[0,:] = mcolors.to_rgba('DarkGrey')
vals[1,:] = mcolors.to_rgba('lime')
CMAP_MORPH = mcolors.ListedColormap(vals)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/neatdend/checkouts/latest/examples/
↪bac_firing.py:23: UserWarning: NEURON not available, plotting stored image
  warnings.warn('NEURON not available, plotting stored image', UserWarning)
```

```
def getCTree(cfit, locs, f_name, recompute_ctree=False, recompute_biophys=False):
    """
    Uses `neat.CompartmentFitter` to derive a `neat.CompartmentTree` for the
    given `locs`. The simplified tree is stored under `f_name`. If the
    simplified tree exists, it is loaded by default in memory (unless
    `recompute_ctree` is ``True``). The impedances for efficient impedance
    matrix evaluation are also stored, and are by default reloaded if they exist
    (unless `recompute_biophys` is ``True``).
    """
    try:
```

(continues on next page)

(continued from previous page)

```
        if recompute_ctree:
            raise IOError
        print('\n>>>> loading file %s'%f_name)
        file = open(f_name + '.p', 'rb')
        ctree = pickle.load(file)
        clocs = pickle.load(file)
    except (IOError, EOFError) as err:
        print('\n>>>> (re-)deriving model %s'%f_name)
        ctree = cfit.fitModel(locs, alpha_inds=[0], parallel=True,
                                    use_all_channels_for_passive=False,
                                    recompute=recompute_biophys)
        clocs = ctree.getEquivalentLocs()
        print('>>>> writing file %s'%f_name)
        file = open(f_name + '.p', 'wb')
        pickle.dump(ctree, file)
        pickle.dump(clocs, file)
    file.close()
    return ctree, clocs


def runCaCoinc(sim_tree, locs,
               ca_loc_ind, soma_ind,
               stim_type='psp',
               dt=0.1, t_max=300., t_calibrate=100.,
               psp_params=dict(t_rise=.5, t_decay=5., i_amp=.5, t_stim=50.),
               i_in_params=dict(i_amp=1.9, t_onset=45., t_dur=5.),
               rec_kwargs=dict(record_from_syns=False, record_from_iclamps=False,
                               record_from_vclamps=False, record_from_channels=False,
                               record_v_deriv=False),
               pprint=True):
    """
    Runs the BAC-firing protocol to elicit an AP burst in response to coincident
    somatic and dendritic input
    """
    # initialize the NEURON model
    sim_tree.initModel(dt=dt, t_calibrate=t_calibrate, factor_lambda=10.)
    sim_tree.storeLocs(locs, 'rec locs')
    if stim_type == 'psp' or stim_type == 'coinc':
        sim_tree.addDoubleExpCurrent(locs[ca_loc_ind], psp_params['t_rise'], psp_
↪params['t_decay'])
        sim_tree.setSpikeTrain(0, psp_params['i_amp'], [psp_params['t_stim']])
    if stim_type == 'psp':
        sim_tree.addIClamp(locs[soma_ind], 0., i_in_params['t_onset'], i_in_params['t_
↪dur'])
    if stim_type == 'current':
        sim_tree.addDoubleExpCurrent(locs[ca_loc_ind], psp_params['t_rise'], psp_
↪params['t_decay'])
        sim_tree.setSpikeTrain(0, 0., [psp_params['t_stim']])
    if stim_type == 'current' or stim_type == 'coinc':
        sim_tree.addIClamp(locs[soma_ind], i_in_params['i_amp'], i_in_params['t_onset
↪'], i_in_params['t_dur'])
    # simulate the NEURON model
    res = sim_tree.run(t_max, pprint=pprint, **rec_kwargs)
    sim_tree.deleteModel()
    return res
```

(continues on next page)

(continued from previous page)

```
def runCalciumCoinc(recompute_ctree=False, recompute_biophys=False, axdict=None,␣
↪pshow=True):
    global D2S_CASPIKE, D2S_APIC
    global CA_LOC

    lss_ = ['-', '-.', '--']
    css_ = [colours[3], colours[0], colours[1]]
    lws_ = [.8, 1.2, 1.6]

    # create the full model
    phys_tree = getL5Pyramid()
    sim_tree = phys_tree.__copy__(new_tree=NeuronSimTree())
    # compartmentfitter object
    cfit = CompartmentFitter(phys_tree, name='bac_firing', path='data/')

    # single branch initiation zone
    branch = sim_tree.pathToRoot(sim_tree[236])[::-1]
    locs_sb = sim_tree.distributeLocsOnNodes(D2S_CASPIKE, node_arg=branch, name=
↪'single branch')
    # abpical trunk locations
    apic = sim_tree.pathToRoot(sim_tree[221])[::-1]
    locs_apic = sim_tree.distributeLocsOnNodes(D2S_APIC, node_arg=apic, name='apic␣
↪connection')

    # store set of locations
    fit_locs = [(1, .5)] + locs_apic + locs_sb
    sim_tree.storeLocs(fit_locs, name='ca coinc')
    # PSP input location index
    ca_ind = sim_tree.getNearestLocinds([CA_LOC], name='ca coinc')[0]

    # obtain the simplified tree
    ctree, clocs = getCTree(cfit, fit_locs, 'data/ctree_bac_firing',
            recompute_biophys=recompute_biophys, recompute_ctree=recompute_ctree)

    # print(ctree)
    print('--- ctree nodes currents')
    print('\n'.join([str(n.currents) for n in ctree]))

    reslist, creslist_sb, creslist_sb_ = [], [], []
    locindslist_sb, locindslist_apic_sb = [], []

    if axdict is None:
        pl.figure('inp')
        axes_input = [pl.subplot(131), pl.subplot(132), pl.subplot(133)]
        pl.figure('V trace')
        axes_trace = [pl.subplot(131), pl.subplot(132), pl.subplot(133)]
        pl.figure('morph')
        axes_morph = [pl.subplot(121), pl.subplot(122)]
    else:
        axes_input = axdict['inp']
        axes_trace = axdict['trace']
        axes_morph = axdict['morph']
        pshow = False

    for jj, stim in enumerate(['current', 'psp', 'coinc']):
        print('--- sim full  ---')
        rec_locs = sim_tree.getLocs('ca coinc')
```

(continues on next page)

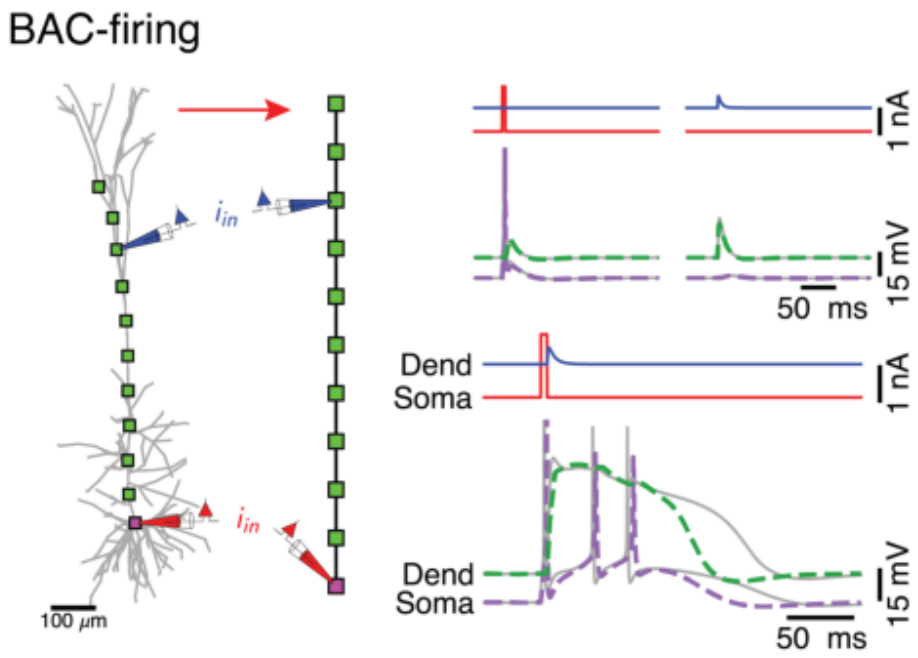
(continued from previous page)

```
        # runn the simulation
        res = runCaCoinc(sim_tree, rec_locs, ca_ind, 0, stim_type=stim,
                    rec_kwargs=dict(record_from_syns=True, record_from_iclamps=True))

        print('---- sim reduced ----')
        rec_locs = clocs
        # run the simulation of the reduced tree
        csim_tree = createReducedNeuronModel(ctree)
        cres = runCaCoinc(csim_tree, rec_locs, ca_ind, 0, stim_type=stim, rec_
↪kwargs=dict(record_from_syns=True, record_from_iclamps=True))

        id_offset = 1.
        vd_offset = 7.2
        vlim = (-80.,20.)
        ilim = (-.1,2.2)

        # input current
        ax = axes_input[jj]
        ax.plot(res['t'], -res['i_clamp'][0], c='r', lw=lwidth)
        ax.plot(res['t'], res['i_syn'][0]+id_offset, c='b', lw=lwidth)

        ax.set_yticks([0., id_offset])
        if jj == 1 or jj == 2:
            drawScaleBars(ax, ylabel=' nA', b_offset=0)
        else:
            drawScaleBars(ax)
        if jj == 2:
            ax.set_yticklabels([r'Soma', r'Dend'])

        ax.set_ylim(ilim)

        # somatic trace
        ax = axes_trace[jj]
        ax.set_xticks([0.,50.])
        ax.plot(res['t'], res['v_m'][0], c='DarkGrey', lw=lwidth)
        ax.plot(cres['t'], cres['v_m'][0], c=cll[0], lw=1.6*lwidth, ls='--')

        # dendritic trace
        ax.plot(res['t'], res['v_m'][ca_ind]+vd_offset, c='DarkGrey', lw=lwidth)
        ax.plot(cres['t'], cres['v_m'][ca_ind]+vd_offset, c=cll[1], lw=1.6*lwidth, ls=
↪'--')

        ax.set_yticks([cres['v_m'][0][0], cres['v_m'][ca_ind][0]+vd_offset])
        if jj == 1 or jj == 2:
            drawScaleBars(ax, xlabel=' ms', ylabel=' mV', b_offset=15)
            # drawScaleBars(ax, xlabel=' ms', b_offset=25)
        else:
            drawScaleBars(ax)
        if jj == 2:
            ax.set_yticklabels([r'Soma', r'Dend'])
        ax.set_ylim(vlim)

    print('iv')

    plocs = sim_tree.getLocs('ca coinc')
    markers = [{'marker': 's', 'mfc': cfl[0], 'mec': 'k', 'ms': markersize/1.1}] + \
              [{'marker': 's', 'mfc': cfl[1], 'mec': 'k', 'ms': markersize/1.1} for _␣
↪in locs_apic + locs_sb]
```

(continues on next page)

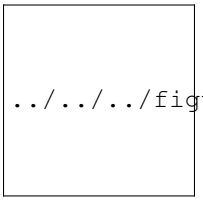
(continued from previous page)

```
    markers[ca_ind]['marker'] = 'v'
    plotargs = {'lw': lwidth/1.3, 'c': 'DarkGrey'}
    sim_tree.plot2DMorphology(axes_morph[0], use_radius=False, plotargs=plotargs,
                              marklocs=plocs, locargs=markers, lims_margin=0.01)
    # compartment tree dendrogram
    labelargs = {0: {'marker': 's', 'mfc': cfl[0], 'mec': 'k', 'ms': markersize*1.2}}
    labelargs.update({ii: {'marker': 's', 'mfc': cfl[1], 'mec': 'k', 'ms':␣
↪markersize*1.2} for ii in range(1,len(plocs))})
    ctree.plotDendrogram(axes_morph[1], plotargs={'c':'k', 'lw': lwidth},␣
↪labelargs=labelargs)

    pl.show()


if __name__ == '__main__':
    if SIM_FLAG:
        runCalciumCoinc()
    else:
        plotStoredImg('../docs/figures/bac_firing.png')
```



Total running time of the script: (0 minutes 0.427 seconds)

6.10.2 Sequence discrimination




```
import sys
import numpy as np
import warnings

from neat import MorphLoc, CompartmentFitter
from models.L23_pyramid import getL23PyramidPas

from plotutil import *

SIM_FLAG = 1
try:
    import neuron
    from neuron import h
    from neat import NeuronSimTree, NeuronCompartmentTree, createReducedNeuronModel
except ImportError:
    warnings.warn('NEURON not available, plotting stored image', UserWarning)
    SIM_FLAG = 0


## Parameters #########################################################################
# synapse location parameters
SYN_NODE_IND = 112 # node index corresponding to dend[13] in Branco's (2010) model
SYN_XCOMP = [0., 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.]
# AMPA synapse parameters
G_MAX_AMPA = 0.0005 # 500 pS
TAU_AMPA = 2.       # ms
# NMDA synapse parameters
G_MAX_NMDA = 8000.  # 8000 pS (Popen is 0.2 so effective gmax = 1600 pS, use 5000 pS␣
↪for active model)
E_REV_NMDA = 5.     # mV
C_MG = 1.           # mM
DUR_REL = 0.5       # ms
AMP_REL = 2.        # mM
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/neatdend/checkouts/latest/examples/
↪sequence_discrimination.py:23: UserWarning: NEURON not available, plotting stored␣
↪image
  warnings.warn('NEURON not available, plotting stored image', UserWarning)
```

```
def plotSim(delta_ts=[0.,1.,2.,3.,4.,5.,6.,7.,8.], recompute=False):
    class BrancoSimTree(NeuronSimTree):
        '''
        Inherits from :class:`NeuronSimTree` to implement Branco model
        '''
        def __init__(self):
```

(continues on next page)

(continued from previous page)

```
            super().__init__()
            phys_tree = getL23PyramidPas()
            phys_tree.__copy__(new_tree=self)

        def setSynLocs(self):
            global SYN_NODE_IND, SYN_XCOMP
            # set computational tree
            self.setCompTree()
            self.treetype = 'computational'
            # define the locations
            locs = [MorphLoc((SYN_NODE_IND, x), self, set_as_comploc=True) for x in␣
↪SYN_XCOMP]
            self.storeLocs(locs, name='syn locs')
            self.storeLocs([(1., 0.5)], name='soma loc')
            # set treetype back
            self.treetype = 'original'

        def deleteModel(self):
            super(BrancoSimTree, self).deleteModel()
            self.pres = []
            self.nmdas = []

        def addAMPASynapse(self, loc, g_max, tau):
            loc = MorphLoc(loc, self)
            # create the synapse
            syn = h.AlphaSynapse(self.sections[loc['node']](loc['x']))
            syn.tau = tau
            syn.gmax = g_max
            # store the synapse
            self.syns.append(syn)

        def addNMDASynapse(self, loc, g_max, e_rev, c_mg, dur_rel, amp_rel):
            loc = MorphLoc(loc, self)
            # create the synapse
            syn = h.NMDA_Mg_T(self.sections[loc['node']](loc['x']))
            syn.gmax = g_max
            syn.Erev = e_rev
            syn.mg = c_mg
            # create the presynaptic segment for release
            pre = h.Section(name='pre %d'%len(self.pres))
            pre.insert('release_BMK')
            pre(0.5).release_BMK.dur = dur_rel
            pre(0.5).release_BMK.amp = amp_rel
            # connect
            h.setpointer(pre(0.5).release_BMK._ref_T, 'C', syn)
            # store the synapse
            self.nmdas.append(syn)
            self.pres.append(pre)

            # setpointer cNMDA[n].C, PRE[n].T_rel(0.5)
            # setpointer im_xtra(x), i_membrane(x)
            # h.setpointer(dend(seg.x)._ref_i_membrane, 'im', dend(seg.x).xtra)

        def setSpikeTime(self, syn_index, spike_time):
            spk_tm = spike_time + self.t_calibrate
            # add spike for AMPA synapse
            self.syns[syn_index].onset = spk_tm
```

(continues on next page)

(continued from previous page)

```
            # add spike for NMDA synapse
            self.pres[syn_index](0.5).release_BMK.delay = spk_tm

        def addAllSynapses(self):
            global G_MAX_AMPA, TAU_AMPA, G_MAX_NMDA, E_REV_NMDA, C_MG, DUR_REL, AMP_
↪REL
            for loc in self.getLocs('syn locs'):
                # ampa synapse
                self.addAMPASynapse(loc, G_MAX_AMPA, TAU_AMPA)
                # nmda synapse
                self.addNMDASynapse(loc, G_MAX_NMDA, E_REV_NMDA, C_MG, DUR_REL, AMP_
↪REL)

        def setSequence(self, delta_t, centri='fugal', t0=10., tadd=100.):
            n_loc = len(self.getLocs('syn locs'))
            if centri == 'fugal':
                for ll in range(n_loc):
                    self.setSpikeTime(ll, t0 + ll * delta_t)
            elif centri == 'petal':
                for tt, ll in enumerate(range(n_loc)[::-1]):
                    self.setSpikeTime(ll, t0 + tt * delta_t)
            else:
                raise IOError('Only centrifugal or centripetal sequences are allowed,
↪' + \
                              'use \'fugal\' resp. \'petal\' as second arg.')
            return n_loc * delta_t + t0 + tadd

        def reduceModel(self, pprint=False):
            global SYN_NODE_IND, SYN_XCOMP
            locs = [MorphLoc((1, .5), self, set_as_comploc=True)] + \
                   [MorphLoc((SYN_NODE_IND, x), self, set_as_comploc=True) for x in␣
↪SYN_XCOMP]

            # creat the reduced compartment tree
            ctree = self.createCompartmentTree(locs)
            # create trees to derive fitting matrices
            sov_tree, greens_tree = self.getZTrees()

            # compute the steady state impedance matrix
            z_mat = greens_tree.calcImpedanceMatrix(locs)[0].real
            # fit the conductances to steady state impedance matrix
            ctree.computeGMC(z_mat, channel_names=['L'])

            if pprint:
                np.set_printoptions(precision=1, linewidth=200)
                print(('Zmat original (MOhm) =\n' + str(z_mat)))
                print(('Zmat fitted (MOhm) =\n' + str(ctree.calcImpedanceMatrix())))

            # get SOV constants
            alphas, phimat = sov_tree.getImportantModes(locarg=locs,
                                                        sort_type='importance',␣
↪eps=1e-12)
            # n_mode = len(locs)
            # alphas, phimat = alphas[:n_mode], phimat[:n_mode, :]
            importance = sov_tree.getModeImportance(sov_data=(alphas, phimat),␣
↪importance_type='full')
            # fit the capacitances from SOV time-scales
```

(continues on next page)

(continued from previous page)

```
            # ctree.computeC(-alphas*1e3, phimat, weight=importance)
            ctree.computeC(-alphas[:1]*1e3, phimat[:1,:], importance=importance[:1])

            if pprint:
                print(('Taus original (ms) =\n' + str(np.abs(1./alphas))))
                lambdas, _, _ = ctree.calcEigenvalues()
                print(('Taus fitted (ms) =\n' + str(np.abs(1./lambdas))))

            return ctree

        def runSim(self, delta_t=12.):
            try:
                el = self[0].currents['L'][1]
            except AttributeError:
                el = self[1].currents['L'][1]
            # el=-75.

            self.initModel(dt=0.025, t_calibrate=0., v_init=el, factor_lambda=10.)
            # add the synapses
            self.addAllSynapses()
            t_max = self.setSequence(delta_t, centri='petal')
            # set recording locs
            self.storeLocs(self.getLocs('soma loc') + self.getLocs('syn locs'), name=
↪'rec locs')
            # run the simulation
            res_centripetal = self.run(t_max, pprint=True)
            # delete the model
            self.deleteModel()

            self.initModel(dt=0.025, t_calibrate=0., v_init=el, factor_lambda=10.)
            # add the synapses
            self.addAllSynapses()
            t_max = self.setSequence(delta_t, centri='fugal')
            # set recording locs
            self.storeLocs(self.getLocs('soma loc') + self.getLocs('syn locs'), name=
↪'rec locs')
            # run the simulation
            res_centrifugal = self.run(t_max, pprint=True)
            # delete the model
            self.deleteModel()

            return res_centripetal, res_centrifugal


    class BrancoReducedTree(NeuronCompartmentTree, BrancoSimTree):
        def __init__(self):
            # call the initializer of :class:`NeuronSimTree`, follows after
            # :class:`BrancoSimTree` in MRO
            super(BrancoSimTree, self).__init__(file_n=None, types=[1,3,4])

        def setSynLocs(self, equivalent_locs):
            self.storeLocs(equivalent_locs[1:], name='syn locs')
            self.storeLocs(equivalent_locs[:1], name='soma loc')

    global SYN_NODE_IND, SYN_XCOMP

    # initialize the full model
```

(continues on next page)

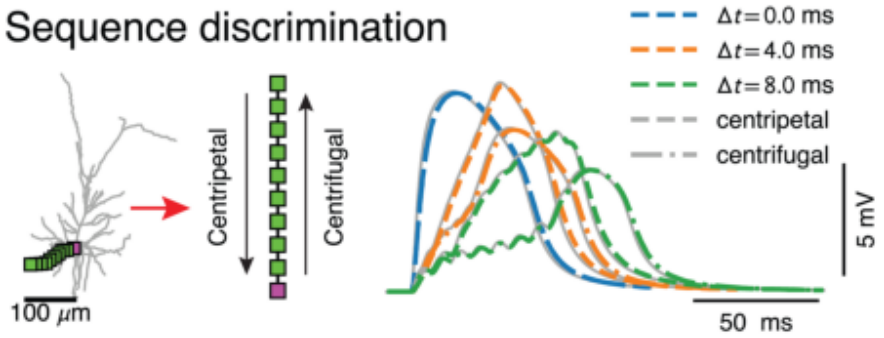
(continued from previous page)

```
    simtree = BrancoSimTree()
    simtree.setSynLocs()
    simtree.setCompTree()

    # derive the reduced model retaining only soma and synapse locations
    fit_locs = simtree.getLocs('soma loc') + simtree.getLocs('syn locs')
    c_fit = CompartmentFitter(simtree, name='sequence_discrimination', path='data/')
    ctree = c_fit.fitModel(fit_locs, recompute=recompute)
    clocs = ctree.getEquivalentLocs()

    # create the reduced model for NEURON simulation
    csimtree_ = createReducedNeuronModel(ctree)
    csimtree = csimtree_.__copy__(new_tree=BrancoReducedTree())
    csimtree.setSynLocs(clocs)

    pl.figure('Branco', figsize=(5,5))
    gs = GridSpec(2,2)
    ax0 = pl.subplot(gs[0,0])
    ax_ = pl.subplot(gs[0,1])
    ax1 = myAx(pl.subplot(gs[1,:]))

    # plot the full morphology
    locargs = [dict(marker='s', mec='k', mfc=cfl[0], ms=markersize)]
    locargs.extend([dict(marker='s', mec='k', mfc=cfl[1], ms=markersize) for ii in␣
↪range(1,len(fit_locs))])
    pnodes = [n for n in simtree if n.swc_type != 2]
    plotargs = {'lw': lwidth/1.3, 'c': 'DarkGrey'}
    simtree.plot2DMorphology(ax0, use_radius=False,node_arg=pnodes,
                             plotargs=plotargs, marklocs=fit_locs, locargs=locargs,␣
↪lims_margin=.01,
                             textargs={'fontsize': ticksize}, labelargs={'fontsize':␣
↪ticksize})

    # plot a schematic of the reduced model
    labelargs = {0: {'marker': 's', 'mfc': cfl[0], 'mec': 'k', 'ms': markersize*1.2}}
    labelargs.update({ii: {'marker': 's', 'mfc': cfl[1], 'mec': 'k', 'ms':␣
↪markersize*1.2} for ii in range(1,len(fit_locs))})
    ctree.plotDendrogram(ax_, plotargs={'c':'k', 'lw': lwidth}, labelargs=labelargs)

    xlim, ylim = np.array(ax_.get_xlim()), np.array(ax_.get_ylim())
    pp = np.array([np.mean(xlim), np.mean(ylim)])
    dp = np.array([2.*np.abs(xlim[1]-xlim[0])/3.,0.])
    ax_.annotate('Centrifugal', #xycoords='axes points',
                 xy=pp, xytext=pp+dp,
                 size=ticksize, rotation=90, ha='center', va='center')
    ax_.annotate('Centripetal', #xycoords='axes points',
                 xy=pp, xytext=pp-dp,
                 size=ticksize, rotation=90, ha='center', va='center')

    # plot voltage traces
    legend_handles = []
    for ii, delta_t in enumerate(delta_ts):
        res_cp, res_cf = simtree.runSim(delta_t=delta_t)
        ax1.plot(res_cp['t'], res_cp['v_m'][0], c='DarkGrey', lw=lwidth)
        ax1.plot(res_cf['t'], res_cf['v_m'][0], c='DarkGrey', lw=lwidth)

        cres_cp, cres_cf = csimtree.runSim(delta_t=delta_t)
```

(continues on next page)

(continued from previous page)

```
        line = ax1.plot(cres_cp['t'], cres_cp['v_m'][0], c=colours[ii%len(colours)],
                        ls='--', lw=1.6*lwidth, label=r'$\Delta t = ' + '%.1f$ ms'
↪%delta_t)
        ax1.plot(cres_cf['t'], cres_cf['v_m'][0], c=colours[ii%len(colours)], ls='-.',
↪ lw=1.6*lwidth)

        legend_handles.append(line[0])

    legend_handles.append(mlines.Line2D([0,0],[0,0], ls='--', lw=1.6*lwidth, c=
↪'DarkGrey', label=r'centripetal'))
    legend_handles.append(mlines.Line2D([0,0],[0,0], ls='-.', lw=1.6*lwidth, c=
↪'DarkGrey', label=r'centrifugal'))

    drawScaleBars(ax1, r' ms', r' mV', b_offset=20, h_offset=15, v_offset=15)
    myLegend(ax1, loc='upper left', bbox_to_anchor=(.7,1.3), handles=legend_handles,
                  fontsize=ticksize, handlelength=2.7)

    pl.tight_layout()
    pl.show()


if __name__ == '__main__':
    if SIM_FLAG:
        plotSim(delta_ts=[0.,4.,8.])
    else:
        plotStoredImg('../docs/figures/sequence_discrimination.png')
```



Total running time of the script: (0 minutes 0.206 seconds)

6.10.3 Basal AP Backprop

../../../figures/ap_backpropagation.png

```
import numpy as np
import warnings

from neat import MorphLoc, CompartmentFitter
from models.L23_pyramid import getL23PyramidNaK

from plotutil import *

import pickle

SIM_FLAG = 1
try:
```

(continues on next page)

(continued from previous page)

```
    import neuron
    from neat import NeuronSimTree, NeuronCompartmentTree, createReducedNeuronModel
except ImportError:
    warnings.warn('NEURON not available, plotting stored image', UserWarning)
    SIM_FLAG = 0

## Parameters ##################################################################
# soma nodes branco
SLOCS = [(1, .5)]
# loc params
D2S_BASAL = np.array([50., 100., 150.])
# soma stimulus params
STIM_PARAMS = {'amp': 3., # nA
                      't_onset': 5., # ms
                      't_dur': 1. # ms
                     }
# simulation parameters
DT = 0.025
T_MAX = 300.
TC = 200.

# morphology color map
vals = np.ones((2, 4))
vals[0,:] = mcolors.to_rgba('DarkGrey')
vals[1,:] = mcolors.to_rgba('lime')
CMAP_MORPH = mcolors.ListedColormap(vals)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/neatdend/checkouts/latest/examples/
↪basal_ap_backprop.py:23: UserWarning: NEURON not available, plotting stored image
  warnings.warn('NEURON not available, plotting stored image', UserWarning)
```

```
def getCTree(cfit, locs, f_name, recompute_ctree=False, recompute_biophys=False):
    """
    Uses `neat.CompartmentFitter` to derive a `neat.CompartmentTree` for the
    given `locs`. The simplified tree is stored under `f_name`. If the
    simplified tree exists, it is loaded by default in memory (unless
    `recompute_ctree` is ``True``). The impedances for efficient impedance
    matrix evaluation are also stored, and are by default reloaded if they exist
    (unless `recompute_biophys` is ``True``).
    """
    try:
        if recompute_ctree:
            raise IOError
        print('\n>>>> loading file %s'%f_name)
        file = open(f_name + '.p', 'rb')
        ctree = pickle.load(file)
        clocs = pickle.load(file)
    except (IOError, EOFError) as err:
        print('\n>>>> (re-)deriving model %s'%f_name)
        ctree = cfit.fitModel(locs, alpha_inds=[0], parallel=True,
                                    use_all_channels_for_passive=False,
                                    recompute=recompute_biophys)
        clocs = ctree.getEquivalentLocs()
        print('>>>> writing file %s'%f_name)
```

(continues on next page)

(continued from previous page)

```
        file = open(f_name + '.p', 'wb')
        pickle.dump(ctree, file)
        pickle.dump(clocs, file)
    file.close()
    return ctree, clocs


def calcAmpDelayWidth(res):
    """
    Compute a number of AP amplitude, delay compared to start of simulation,
    delay of backpropagating AP compared to soma AP, and halfwidth
    """
    dt = res['t'][1] - res['t'][0]
    # amplitude of peak
    res['amp'] = np.max(res['v_m'], axis=1) - res['v_m'][:,0]
    # delay of peak compared to soma
    res['delay'] = dt * (np.argmax(res['v_m'], axis=1) - np.argmax(res['v_m'][0]))
    # absolute delay of peak
    res['dop'] = dt * np.argmax(res['v_m'], axis=1)
    # width of waveform at half amplitude
    v_half = res['amp'] / 2. + res['v_m'][:,0]
    res['width'] = dt*np.sum(res['v_m'] > v_half[:,None], axis=1)


def runSim(simtree, locs, soma_loc, stim_params={'amp':.5, 't_onset':5., 't_dur':1.}):
    """
    Runs simulation to inject somatic current in order to elicit AP
    """
    global DT, T_MAX, TC
    global T_DUR, G_SYN, N_INP

    simtree.initModel(dt=DT, t_calibrate=TC, factor_lambda=1.)
    simtree.addIClamp(soma_loc, stim_params['amp'], stim_params['t_onset'], stim_
↪params['t_dur'])
    simtree.storeLocs([soma_loc] + locs, 'rec locs')

    res = simtree.run(40., record_from_iclamps=True)
    simtree.deleteModel()

    return res


def basalAPBackProp(recompute_ctree=False, recompute_biophys=False, axes=None,␣
↪pshow=True):
    global STIM_PARAMS, D2S_BASAL, SLOCS
    global CMAP_MORPH

    rc, rb  = recompute_ctree, recompute_biophys

    if axes is None:
        pl.figure(figsize=(7,5))
        ax1, ax2, ax4, ax5 = pl.subplot(221), pl.subplot(223), pl.subplot(222), pl.
↪subplot(224)
        divider = make_axes_locatable(ax1)
        ax3 =  divider.append_axes("top", "30%", pad="10%")
        ax4, ax5 = myAx(ax4), myAx(ax5)
        pl.figure(figsize=(5,5))
```

(continues on next page)

(continued from previous page)

```
        gs = GridSpec(2,2)
        ax_morph, ax_red1, ax_red2 = pl.subplot(gs[0,:]), pl.subplot(gs[1,0]), pl.
↪subplot(gs[1,1])
    else:
        ax1, ax2, ax3 = axes['trace']
        ax4, ax5 = axes['amp-delay']
        ax_morph, ax_red1, ax_red2 = axes['morph']
        pshow = False

    # create the full model
    phys_tree = getL23PyramidNaK()
    sim_tree = phys_tree.__copy__(new_tree=NeuronSimTree())

    # distribute locations to measure backAPs on branches
    leafs_basal = [node for node in sim_tree.leafs if node.swc_type == 3]
    branches    = [sim_tree.pathToRoot(leaf)[::-1] for leaf in leafs_basal]
    locslist    = [sim_tree.distributeLocsOnNodes(D2S_BASAL, node_arg=branch) for␣
↪branch in branches]
    branchlist  = [b for ii, b in enumerate(branches) if len(locslist[ii]) == 3]
    locs    = [locs for locs in locslist if len(locs) == 3][1]
    # do back prop sims
    amp_diffs_3loc, delay_diffs_3loc = np.zeros(3), np.zeros(3)
    amp_diffs_1loc, delay_diffs_1loc = np.zeros(3), np.zeros(3)
    amp_diffs_biop, delay_diffs_biop = np.zeros(3), np.zeros(3)

    # compartmentfitter object
    cfit = CompartmentFitter(phys_tree, name='basal_bAP', path='data/')

    # create reduced tree
    ctree, clocs = getCTree(cfit, [SLOCS[0]] + locs, 'data/ctree_basal_bAP_3loc',
                            recompute_ctree=rc, recompute_biophys=rb)
    csimtree = createReducedNeuronModel(ctree)
    print(ctree)

    # run the simulation of he full tree
    res = runSim(sim_tree, locs, SLOCS[0], stim_params=STIM_PARAMS)
    calcAmpDelayWidth(res)

    amp_diffs_biop[:] = res['amp'][1:]
    delay_diffs_biop[:] = res['delay'][1:]

    # run the simulation of the reduced tree
    cres = runSim(csimtree, clocs[1:], clocs[0], stim_params=STIM_PARAMS)
    calcAmpDelayWidth(cres)

    amp_diffs_3loc[:] = cres['amp'][1:]
    delay_diffs_3loc[:] = cres['delay'][1:]

    # reduced models with one single dendritic site
    creslist = []
    for jj, loc in enumerate(locs):
        # create reduced tree with all 1 single dendritic site locs
        ctree, clocs = getCTree(cfit, [SLOCS[0]] + [loc], 'data/ctree_basal_bAP_1loc%d
↪'%jj,
                                recompute_ctree=rc, recompute_biophys=False)
        csimtree = createReducedNeuronModel(ctree)
        print(ctree)
```

(continues on next page)

(continued from previous page)

```
        # run the simulation of the reduced tree
        cres_ss = runSim(csimtree, [clocs[1]], clocs[0], stim_params=STIM_PARAMS)
        calcAmpDelayWidth(cres_ss)
        creslist.append(cres_ss)

        amp_diffs_1loc[jj] = cres_ss['amp'][1]
        delay_diffs_1loc[jj] = cres_ss['delay'][1]


    ylim = (-90., 60.)
    x_range = np.array([-3.,14])
    xlim = (0., 12.)

    tp_full = res['t'][np.argmax(res['v_m'][0])]
    tp_3comp = cres['t'][np.argmax(cres['v_m'][0])]
    tp_1comp = creslist[2]['t'][np.argmax(creslist[2]['v_m'][0])]

    tlim_full = tp_full + x_range
    tlim_3comp = tp_3comp + x_range
    tlim_1comp = tp_1comp + x_range

    i0_full, i1_full = np.round(tlim_full / DT).astype(int)
    i0_3comp, i1_3comp = np.round(tlim_3comp / DT).astype(int)
    i0_1comp, i1_1comp = np.round(tlim_1comp / DT).astype(int)

    ax1.set_ylabel(r'soma')
    ax1.plot(res['t'][i0_full:i1_full] - tlim_full[0], res['v_m'][0][i0_full:i1_full],
             lw=lwidth, c='DarkGrey', label=r'full')
    ax1.plot(cres['t'][i0_3comp:i1_3comp] - tlim_3comp[0], cres['v_m'][0][i0_3comp:i1_
↪3comp],
             ls='--', lw=1.6*lwidth, c=colours[0], label=r'3 comp')
    ax1.plot(creslist[2]['t'][i0_1comp:i1_1comp] - tlim_1comp[0], creslist[2]['v_m
↪'][0][i0_1comp:i1_1comp],
             ls='-.', lw=1.6*lwidth, c=colours[1], label=r'1 comp')

    ax1.set_ylim(ylim)
    # ax1.set_xlim(xlim)
    drawScaleBars(ax1, b_offset=15)

    myLegend(ax1, add_frame=False, loc='center left', bbox_to_anchor=[0.35, 0.55],␣
↪fontsize=ticksize,
                  labelspacing=.8, handlelength=2., handletextpad=.2)

    ax2.set_ylabel(r'dend' + '\n($d_{soma} = 150$ $\mu$m)')
    ax2.plot(res['t'][i0_full:i1_full] - tlim_full[0], res['v_m'][3][i0_full:i1_full],
             lw=lwidth, c='DarkGrey', label=r'full')
    ax2.plot(cres['t'][i0_3comp:i1_3comp] - tlim_3comp[0], cres['v_m'][3][i0_3comp:i1_
↪3comp],
             ls='--', lw=1.6*lwidth, c=colours[0], label=r'3 comp')
    ax2.plot(creslist[2]['t'][i0_1comp:i1_1comp] - tlim_1comp[0], creslist[2]['v_m
↪'][1][i0_1comp:i1_1comp],
             ls='-.', lw=1.6*lwidth, c=colours[1], label=r'1 comp')

    imax = np.argmax(res['v_m'][3])
    xp = res['t'][imax]
```

(continues on next page)

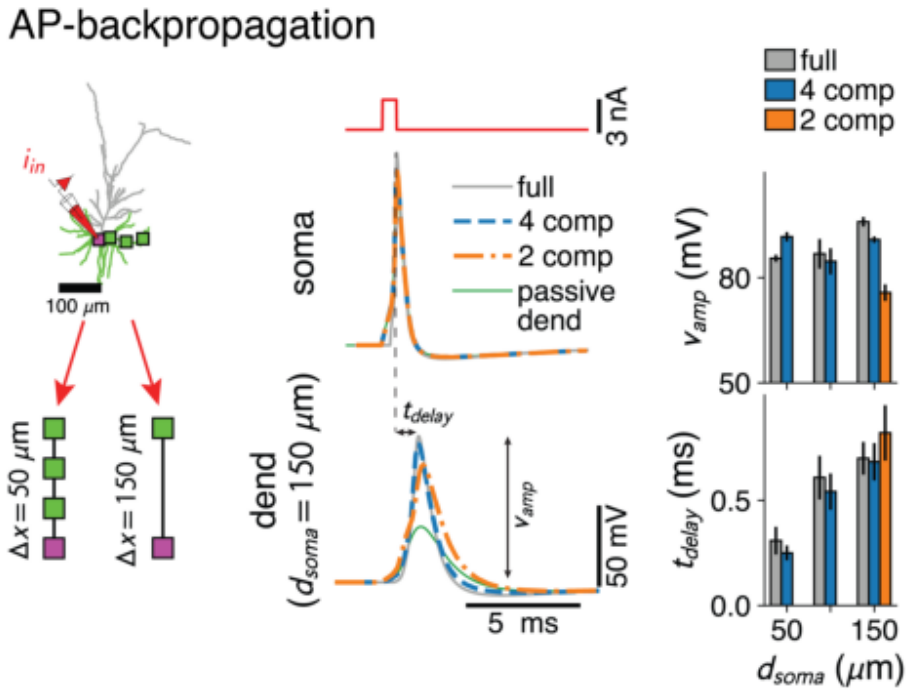
(continued from previous page)

```
    ax2.annotate(r'$v_{amp}$',
                 xy=(xlim[0], np.mean(ylim)), xytext=(xlim[0], np.mean(ylim)),
                 fontsize=ticksize, ha='center', va='center', rotation=90.)
    ax2.annotate(r'$t_{delay}$',
                 xy=(xp, ylim[1]), xytext=(xp, ylim[1]),
                 fontsize=ticksize, ha='center', va='center', rotation=0.)

    ax2.set_ylim(ylim)
    ax2.set_xlim(xlim)

    drawScaleBars(ax2, xlabel=' ms', ylabel=' mV', b_offset=15)

    # myLegend(ax2, add_frame=False, ncol=2, fontsize=ticksize,
    #              loc='upper center', bbox_to_anchor=[.5, -.1],
    #              labelspacing=.6, handlelength=2., handletextpad=.2, columnspacing=.
↪5)

    ax3.plot(res['t'][i0_full:i1_full] - tlim_full[0], -res['i_clamp'][0][i0_full:i1_
↪full],
             lw=lwidth, c='r')
    ax3.set_yticks([0.,3.])
    drawScaleBars(ax3, ylabel=' nA', b_offset=0)
        # ax3.set_xlim(xlim)

    # color the branches
    cnodes = [b for branch in branches for b in branch]
    if cnodes is None:
        plotargs = {'lw': lwidth/1.3, 'c': 'DarkGrey'}
        cs = {node.index: 0 for node in sim_tree}
    else:
        plotargs = {'lw': lwidth/1.3}
        cinds = [n.index for n in cnodes]
        cs = {node.index: 1 if node.index in cinds else 0 for node in sim_tree}
    # mark example locations
    plocs = [SLOCS[0]] + locs
    markers = [{'marker': 's', 'c': cfl[0], 'mec': 'k', 'ms': markersize}] + \
              [{'marker': 's', 'c': cfl[1], 'mec': 'k', 'ms': markersize} for _ in␣
↪plocs[1:]]
    # plot morphology
    sim_tree.plot2DMorphology(ax_morph, use_radius=False, plotargs=plotargs,
                              cs=cs, cmap=CMAP_MORPH,
                              marklocs=plocs, locargs=markers, lims_margin=0.01)

    # plot compartment tree schematic
    ctree_3l = cfit.setCTree([SLOCS[0]] + locs)
    ctree_3l = cfit.ctree
    ctree_1l = cfit.setCTree([SLOCS[0]] + locs[0:1])
    ctree_1l = cfit.ctree

    labelargs = {0: {'marker': 's', 'mfc': cfl[0], 'mec': 'k', 'ms': markersize*1.2}}
    labelargs.update({ii: {'marker': 's', 'mfc': cfl[1], 'mec': 'k', 'ms':␣
↪markersize*1.2} for ii in range(1,len(plocs))})
    ctree_3l.plotDendrogram(ax_red1, plotargs={'c':'k', 'lw': lwidth},␣
↪labelargs=labelargs)

    labelargs = {0: {'marker': 's', 'mfc': cfl[0], 'mec': 'k', 'ms': markersize*1.2},
                 1: {'marker': 's', 'mfc': cfl[1], 'mec': 'k', 'ms': markersize*1.2}}
```

(continues on next page)

(continued from previous page)

```
    ctree_11.plotDendrogram(ax_red2, plotargs={'c':'k', 'lw': lwidth},␣
↪labelargs=labelargs)

    ax_red1.set_xticks([]); ax_red1.set_yticks([])
    ax_red1.set_xlabel(r'$\Delta x = 50$ $\mu$m', fontsize=ticksize,rotation=60)
    ax_red2.set_xticks([]); ax_red2.set_yticks([])
    ax_red2.set_xlabel(r'$\Delta x = 150$ $\mu$m', fontsize=ticksize,rotation=60)

    xb = np.arange(3)
    bwidth = 1./4.
    xtls = [r'50', r'100', r'150']

    ax4, ax5 = myAx(ax4), myAx(ax5)

    ax4.bar(xb-bwidth,        amp_diffs_biop,      width=bwidth, align='center',␣
↪color='DarkGrey', edgecolor='k', label=r'full')
    ax4.bar(xb,               amp_diffs_3loc,      width=bwidth, align='center',␣
↪color=colours[0], edgecolor='k', label=r'4 comp')
    ax4.bar((xb+bwidth)[-1:], amp_diffs_1loc[-1:], width=bwidth, align='center',␣
↪color=colours[1], edgecolor='k', label=r'2 comp')

    ax4.set_ylabel(r'$v_{amp}$ (mV)')
    ax4.set_xticks(xb)
    ax4.set_xticklabels([])
    ax4.set_ylim(50.,110.)
    ax4.set_yticks([50., 80.])

    myLegend(ax4, add_frame=False, loc='lower center', bbox_to_anchor=[.5, 1.05],␣
↪fontsize=ticksize,
                        labelspacing=.1, handlelength=1., handletextpad=.2,␣
↪columnspacing=.5)

    ax5.bar(xb-bwidth,        delay_diffs_biop,      width=bwidth, align='center',␣
↪color='DarkGrey', edgecolor='k', label=r'full')
    ax5.bar(xb,               delay_diffs_3loc,      width=bwidth, align='center',␣
↪color=colours[0], edgecolor='k', label=r'4 comp')
    ax5.bar((xb+bwidth)[-1:], delay_diffs_1loc[-1:], width=bwidth, align='center',␣
↪color=colours[1], edgecolor='k', label=r'2 comp')

    ax5.set_ylabel(r'$t_{delay}$ (ms)')
    ax5.set_xticks(xb)
    ax5.set_xticklabels(xtls)
    ax5.set_xlabel(r'$d_{soma}$ ($\mu$m)')
    ax5.set_yticks([0., 0.5])

    if pshow:
        pl.show()


if __name__ == '__main__':
    if SIM_FLAG:
        basalAPBackProp()
    else:
        plotStoredImg('../docs/figures/ap_backpropagation.png')
```



Total running time of the script: (0 minutes 0.404 seconds)

6.11 Glossary

environment A structure where information about all documents under the root is saved, and used for cross-referencing. The environment is pickled after the parsing stage, so that successive runs only need to read and parse new and changed documents.

source directory The directory which, including its subdirectories, contains all source files for one Sphinx project.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Cannon1998] Cannon et al. (1998) *An online archive of reconstructed hippocampal neurons*, J. Neurosci. methods.

[Carnevale2004] Carnevale, Nicholas T. and Hines, Michael L. (2004) *The NEURON book*

[Koch1985] Koch, C. and Poggio, T. (1985) *A simple algorithm for solving the cable equation in dendritic trees of arbitrary geometry*, Journal of neuroscience methods, 12(4), pp. 303–315.

[Major1993] Major et al. (1993) *Solutions for transients in arbitrarily branching cables: I. Voltage recording with a somatic shunt*, Biophysical journal, 65(1), pp. 423–49.

[Martelli03] A. Martelli (2003) *Python in a Nutshell*, O'Reilly Media Inc.

[Wybo2019] Wybo, Willem A.M. et al. (2019) *Electrical Compartmentalization in Neurons*, Cell Reports, 26(7), pp. 1759–1773

[Wybo2021] Wybo, Willem A.M. et al. (2021) *Data-driven reduction of dendritic morphologies with preserved dendro-somatic responses*, eLife, 10:e60936, pp. 1–26

[Gewaltig2007] Gewaltig, Marc-Oliver and Diesmann, Markus. (2007) *NEST (NEural Simulation Tool)*, Scholarpedia, 2(4), pp. 1430

PYTHON MODULE INDEX

t

neat.trees, 33

INDEX

Symbols

__call__() (*FourrierTools method*), 101
__copy__() (*MorphTree method*), 72
__copy__() (*STree method*), 35
__getitem__() (*MorphTree method*), 56
__getitem__() (*STree method*), 34
__iter__() (*MorphTree method*), 57
__iter__() (*STree method*), 34
__len__() (*STree method*), 34
__str__() (*STree method*), 35
_convertLocArgToLocs() (*MorphTree method*), 62
_convertNodeArgToNodes() (*MorphTree method*), 59
_evaluateCompCriteria() (*MorphTree method*), 60
_evaluateCompCriteria() (*PhysTree method*), 77
_tryName() (*MorphTree method*), 63

A

addCurrent() (*CompartmentTree method*), 43
addCurrent() (*PhysTree method*), 76
addDoubleExpCurrent() (*NeuronSimTree method*), 84
addDoubleExpNMDASynapse() (*NeuronSimTree method*), 85
addDoubleExpSynapse() (*NeuronSimTree method*), 85
addExpSynapse() (*NeuronSimTree method*), 85
addIClamp() (*NeuronSimTree method*), 86
addLoc() (*MorphTree method*), 62
addNMDASynapse() (*NeuronSimTree method*), 85
addNodeWithParent() (*STree method*), 37
addNodeWithParentFromIndex() (*STree method*), 37
addOUClamp() (*NeuronSimTree method*), 86
addOUconductance() (*NeuronSimTree method*), 87
addOUReversal() (*NeuronSimTree method*), 87
addShunt() (*NeuronSimTree method*), 84
addSinClamp() (*NeuronSimTree method*), 86
addVClamp() (*NeuronSimTree method*), 87
asPassiveMembrane() (*PhysTree method*), 74

C

calcConductanceMatrix() (*CompartmentTree method*), 44
calcEEq() (*NeuronSimTree method*), 88
calcEigenvalues() (*CompartmentTree method*), 45
calcImpedanceMatrix() (*CompartmentTree method*), 44
calcImpedanceMatrix() (*GreensTree method*), 81
calcImpedanceMatrix() (*NET method*), 51
calcImpedanceMatrix() (*SOVTree method*), 79
calcImpMat() (*NET method*), 51
calcIZ() (*NET method*), 51
calcIZMatrix() (*NET method*), 51
calcSOVEquations() (*SOVTree method*), 78
calcSystemMatrix() (*CompartmentTree method*), 45
calcTotalImpedance() (*NET method*), 50
calcZF() (*GreensTree method*), 81
checkOrdered() (*STree method*), 35
checkPassive() (*CompartmentFitter method*), 90
child_nodes() (*MorphNode property*), 73
clearLocs() (*MorphTree method*), 62
colorXAxis() (*MorphTree method*), 70
CompartmentFitter (*class in neat*), 89
CompartmentNode (*class in neat*), 48
CompartmentTree (*class in neat*), 42
computeC() (*CompartmentTree method*), 47
computeDerivatives() (*IonChannel method*), 98
computeDerivativesConc() (*IonChannel method*), 99
computeFakeGeometry() (*CompartmentTree method*), 48
computeGChanFromImpedance() (*CompartmentTree method*), 46
computeGMC() (*CompartmentTree method*), 45
computeGSingleChanFromImpedance() (*CompartmentTree method*), 46
computeLinConc() (*IonChannel method*), 100
computeLinear() (*IonChannel method*), 99
computeLinearConc() (*IonChannel method*), 100
computeLinSum() (*IonChannel method*), 100
computeLinTerms() (*SOVTree method*), 80

computePOpen() (*IonChannel method*), 98
computeTauinf() (*IonChannel method*), 99
computeVarinf() (*IonChannel method*), 99
constructNET() (*SOVTree method*), 79
createCompartmentTree() (*MorphTree method*), 72
createNewTree() (*MorphTree method*), 72
createReducedNeuronModel() (*in module neat.tools.simtools.neuron.neuronmodel*), 53
createTreeGF() (*CompartmentFitter method*), 92
createTreeSOV() (*CompartmentFitter method*), 92

D

degreeOfNode() (*STree method*), 38
deleteModel() (*NeuronSimTree method*), 84
depthOfNode() (*STree method*), 38
determineSomaType() (*MorphTree method*), 56
distancesToBifurcation() (*MorphTree method*), 66
distancesToSoma() (*MorphTree method*), 65
distributeLocsOnNodes() (*MorphTree method*), 66
distributeLocsRandom() (*MorphTree method*), 67
distributeLocsUniform() (*MorphTree method*), 66
downBifurcationNode() (*STree method*), 41

E

environment, **155**
evalChannel() (*CompartmentFitter method*), 93
extendWithBifurcationLocs() (*MorphTree method*), 67

F

fitCapacitance() (*CompartmentFitter method*), 94
fitChannels() (*CompartmentFitter method*), 94
fitEEq() (*CompartmentFitter method*), 95
fitEL() (*CompartmentTree method*), 44
fitLeakCurrent() (*PhysTree method*), 76
fitModel() (*CompartmentFitter method*), 90
fitPassive() (*CompartmentFitter method*), 93
fitPassiveLeak() (*CompartmentFitter method*), 93
fitSynRescale() (*CompartmentFitter method*), 95
FourrierTools (*class in neat*), 101
ft() (*FourrierTools method*), 102
ft() (*Kernel method*), 53
ftInv() (*FourrierTools method*), 102

G

gatherNodes() (*STree method*), 36
getBifurcationNodes() (*STree method*), 41
getChannelsInTree() (*PhysTree method*), 76
getCompartmentalization() (*NET method*), 51
getEEq() (*CompartmentFitter method*), 95
getEEq() (*CompartmentTree method*), 43
getEquivalentLocs() (*CompartmentTree method*), 44
getImportantModes() (*SOVTree method*), 78
getKernels() (*CompartmentFitter method*), 91
getLeafLocinds() (*MorphTree method*), 65
getLeafLocNode() (*NET method*), 50
getLeafs() (*MorphTree method*), 58
getLeafs() (*STree method*), 36
getLocInds() (*NET method*), 50
getLocindsOnNode() (*MorphTree method*), 63
getLocindsOnNodes() (*MorphTree method*), 64
getLocindsOnPath() (*MorphTree method*), 64
getLocs() (*MorphTree method*), 63
getModeImportance() (*SOVTree method*), 78
getNearestLocinds() (*MorphTree method*), 64
getNearestNeighbourLocinds() (*MorphTree method*), 65
getNearestNeighbours() (*STree method*), 41
getNodeIndices() (*MorphTree method*), 63
getNodes() (*MorphTree method*), 57
getNodes() (*STree method*), 35
getNodesInApicalSubtree() (*MorphTree method*), 58
getNodesInAxonalSubtree() (*MorphTree method*), 59
getNodesInBasalSubtree() (*MorphTree method*), 58
getNodesInSubtree() (*STree method*), 40
getReducedTree() (*NET method*), 50
getSubTree() (*STree method*), 38
getXCoords() (*MorphTree method*), 63
getXValues() (*MorphTree method*), 69
GreensNode (*class in neat*), 81
GreensTree (*class in neat*), 80

I

initModel() (*NeuronSimTree method*), 84
insertNode() (*STree method*), 38
IonChannel (*class in neat*), 96
isLeaf() (*STree method*), 36
isRoot() (*STree method*), 36

K

k_bar() (*Kernel property*), 53
Kernel (*class in neat*), 52

L

leafs() (*MorphTree property*), 58
leafs() (*STree property*), 36

M

makeXAxis() (*MorphTree method*), 68
module
    neat.trees, 33
MorphLoc (*class in neat*), 73
MorphNode (*class in neat*), 73
MorphTree (*class in neat*), 54

N

neat.trees
    module, 33
NET (*class in neat*), 49
NETNode (*class in neat*), 52
NeuronCompartmentTree (*class in neat.tools.simtools.neuron.neuronmodel*), 53
NeuronSimTree (*class in neat.tools.simtools.neuron.neuronmodel*), 82
nodes() (*MorphTree property*), 58
nodes() (*STree property*), 35

O

orderOfNode() (*STree method*), 39

P

pathBetweenNodes() (*STree method*), 39
pathBetweenNodesDepthFirst() (*STree method*), 39
pathLength() (*MorphTree method*), 68
pathToRoot() (*STree method*), 39
PhysNode (*class in neat*), 77
PhysTree (*class in neat*), 74
plot1D() (*MorphTree method*), 69
plot2DMorphology() (*MorphTree method*), 70
plotDendrogram() (*CompartmentTree method*), 48
plotDendrogram() (*NET method*), 52
plotKernels() (*CompartmentFitter method*), 91
plotMorphologyInteractive() (*MorphTree method*), 71
plotTrueD2S() (*MorphTree method*), 69

R

readSWCTreeFromFile() (*MorphTree method*), 55
removeCompTree() (*MorphTree method*), 60
removeExpansionPoints() (*GreensTree method*), 81
removeLocs() (*MorphTree method*), 62
removeNode() (*STree method*), 37
removeSingleNode() (*STree method*), 37
resetFitData() (*CompartmentTree method*), 47
resetIndices() (*STree method*), 38
root() (*MorphTree property*), 57
root() (*STree property*), 36
run() (*NeuronSimTree method*), 88
runFit() (*CompartmentTree method*), 48

S

setCompTree() (*MorphTree method*), 60
setCTree() (*CompartmentFitter method*), 89
setDefaultParams() (*IonChannel method*), 98
setEEq() (*CompartmentFitter method*), 94
setEEq() (*CompartmentTree method*), 43
setEEq() (*PhysTree method*), 75
setExpansionPoint() (*GreensNode method*), 82
setExpansionPoints() (*CompartmentTree method*), 43
setImpedance() (*GreensTree method*), 81
setLeakCurrent() (*PhysTree method*), 75
setNewLocInds() (*NET method*), 50
setNodeColors() (*MorphTree method*), 69
setP3D() (*MorphNode method*), 73
setPhysiology() (*PhysTree method*), 75
setSpikeTrain() (*NeuronSimTree method*), 87
setTreetype() (*MorphTree method*), 60
sisterLeafs() (*STree method*), 40
SNode (*class in neat*), 41
softRemoveNode() (*STree method*), 37
source directory, **155**
SOVNode (*class in neat*), 80
SOVTree (*class in neat*), 77
storeLocs() (*MorphTree method*), 62
STree (*class in neat*), 33

T

t() (*Kernel method*), 53
treetype() (*MorphTree property*), 60

U

uniqueLocs() (*MorphTree method*), 67
upBifurcationNode() (*STree method*), 40