Dendrify

Release 2.1.2

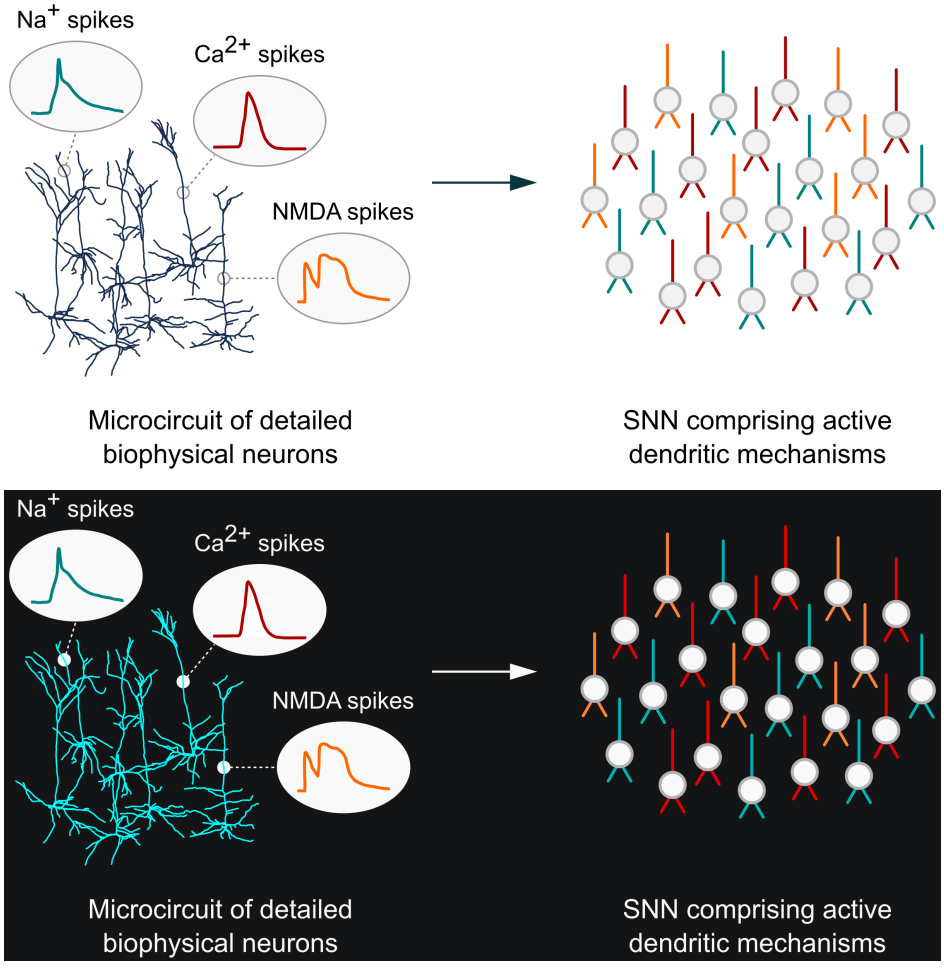
Michalis Pagkalos

Mar 15, 2024

GETTING STARTED

Although neuronal dendrites play a crucial role in shaping how individual neurons process synaptic information, their contribution to network-level functions has remained largely unexplored. Current spiking neural networks (SNNs) often oversimplify dendritic properties or overlook their essential functions. On the other hand, circuit models with morphologically detailed neuron representations are computationally intensive, making them impractical for simulating large networks.

In an effort to bridge this gap, we present Dendrify—a freely available, open-source Python package that seamlessly integrates with the Brian 2 simulator. Dendrify, through simple commands, automatically generates reduced compartmental neuron models with simplified yet biologically relevant dendritic and synaptic integrative properties. These models offer a well-rounded compromise between flexibility, performance, and biological accuracy, enabling us to investigate the impact of dendrites on network-level functions.



Tip: If you use Dendrify for your published research, we kindly ask you to cite our article: **Introducing the Dendrify framework for incorporating dendrites to spiking neural networks** M Pagkalos, S Chavlis, P Poirazi DOI: https://doi.org/10.1038/s41467-022-35747-8

CONTENTS:

CHAPTER

ONE

INSTALLATION

Dendrify is included in the Python package index: https://pypi.org/project/dendrify. The easiest way to install it is through `pip`, using the command:

```
pip install dendrify
```

The above command will automatically install Brian 2.5.4, if it is not already installed. If you wish to work with a different Brian version, you can install Dendrify without its dependencies using the command:

```
pip install dendrify --no-deps
```

and then install the desired version of Brian separately (see bellow).

1.1 Dependencies

- Brian 2 (required) is a simulator for spiking neural networks. It is written in Python and is available on almost all platforms. Brian is designed to be easy to learn and use, highly flexible and easily extensible.
  - Brian 2 installation guidelines
- Networkx (optional) is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. If you wish Dendrify to have access to some experimental model visualization features, you can install it using the command:

  ```
  pip install networkx
  ```

1.2 GPU support

Dendrify is compatible with Brian2CUDA, a Python package for simulating spiking neural networks on graphics processing units (GPUs). Brian2CUDA is an extension of Brian2 that uses the code generation system of the latter to generate simulation code in C++/CUDA, which is then executed on NVIDIA GPUs.

- Brian2CUDA installation guidelines

CHAPTER

TWO

DENDRIFY BASICS

In this tutorial, we are going to cover the following topics:

- Getting to know Dendrify's basic object types and their functions
- How to generate model equations
- How to set model parameters

2.1 Imports

```
import brian2 as b
import dendrify as d
from brian2.units import *
from dendrify import Soma, Dendrite, PointNeuronModel

b.prefs.codegen.target = 'numpy' # faster for simple models and short simulations
```

2.2 Generating model equations

In this first part of the tutorial we are going to focus on how to create single compartments and how to equip them with desired mechanisms.

2.2.1 Creating compartments

```
# Setting a compartment's name is the barely minimum you need to create it
soma = Soma('soma')
dend = Dendrite('dend')
```

```
# Soma and Dendrite objects share many functions since they both inherit from
# the same class
print(isinstance(soma, d.Compartment))
print(isinstance(dend, d.Compartment))
```

```
True
True
```

2.2.2 Accessing equations

```
print(soma.equations)
```

```
dV_soma/dt = (gL_soma * (EL_soma-V_soma) + I_soma) / C_soma  :volt
I_soma = I_ext_soma  :amp
I_ext_soma  :amp
```

```
print(dend.equations)
```

```
dV_dend/dt = (gL_dend * (EL_dend-V_dend) + I_dend) / C_dend  :volt
I_dend = I_ext_dend  :amp
I_ext_dend  :amp
```

2.2.3 Synaptic currents

```
# The usage of tags helps differentiate between same type of synapses that reach
# a single compartment.
dend.synapse('AMPA', tag='A')
```

```
print(dend.equations)
```

```
dV_dend/dt = (gL_dend * (EL_dend-V_dend) + I_dend) / C_dend  :volt
I_dend = I_ext_dend + I_AMPA_A_dend  :amp
I_ext_dend  :amp
I_AMPA_A_dend = g_AMPA_A_dend * (E_AMPA-V_dend) * s_AMPA_A_dend * w_AMPA_A_dend  :amp
ds_AMPA_A_dend/dt = -s_AMPA_A_dend / t_AMPA_decay_A_dend  :1
```

- **s_AMPA_x_dend** -> the state variable for this channel (0 -> closed).
- **w_AMPA_x_dend** -> the weight variable. Useful for plasticity (1 by default).

```
dend.synapse('AMPA', tag='B')
print(dend.equations)
```

```
dV_dend/dt = (gL_dend * (EL_dend-V_dend) + I_dend) / C_dend  :volt
I_dend = I_ext_dend + I_AMPA_B_dend + I_AMPA_A_dend  :amp
I_ext_dend  :amp
I_AMPA_A_dend = g_AMPA_A_dend * (E_AMPA-V_dend) * s_AMPA_A_dend * w_AMPA_A_dend  :amp
ds_AMPA_A_dend/dt = -s_AMPA_A_dend / t_AMPA_decay_A_dend  :1
I_AMPA_B_dend = g_AMPA_B_dend * (E_AMPA-V_dend) * s_AMPA_B_dend * w_AMPA_B_dend  :amp
ds_AMPA_B_dend/dt = -s_AMPA_B_dend / t_AMPA_decay_B_dend  :1
```

```
dend.synapse('NMDA', tag='A')
print(dend.equations)
```

```
dV_dend/dt = (gL_dend * (EL_dend-V_dend) + I_dend) / C_dend  :volt
I_dend = I_ext_dend + I_NMDA_A_dend + I_AMPA_B_dend + I_AMPA_A_dend  :amp
I_ext_dend  :amp
I_AMPA_A_dend = g_AMPA_A_dend * (E_AMPA-V_dend) * s_AMPA_A_dend * w_AMPA_A_dend  :amp
ds_AMPA_A_dend/dt = -s_AMPA_A_dend / t_AMPA_decay_A_dend  :1
I_AMPA_B_dend = g_AMPA_B_dend * (E_AMPA-V_dend) * s_AMPA_B_dend * w_AMPA_B_dend  :amp
ds_AMPA_B_dend/dt = -s_AMPA_B_dend / t_AMPA_decay_B_dend  :1
```

(continues on next page)

(continued from previous page)

```
I_NMDA_A_dend = g_NMDA_A_dend * (E_NMDA-V_dend) * s_NMDA_A_dend / (1 + Mg_con * exp(-
↪Alpha_NMDA*(V_dend/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_A_dend  :amp
ds_NMDA_A_dend/dt = -s_NMDA_A_dend/t_NMDA_decay_A_dend  :1
```

2.2.4 Random noise

```
dend.noise()
print(dend.equations)
```

```
dV_dend/dt = (gL_dend * (EL_dend-V_dend) + I_dend) / C_dend  :volt
I_dend = I_ext_dend + I_noise_dend + I_NMDA_A_dend + I_AMPA_B_dend + I_AMPA_A_dend  :amp
I_ext_dend  :amp
I_AMPA_A_dend = g_AMPA_A_dend * (E_AMPA-V_dend) * s_AMPA_A_dend * w_AMPA_A_dend  :amp
ds_AMPA_A_dend/dt = -s_AMPA_A_dend / t_AMPA_decay_A_dend  :1
I_AMPA_B_dend = g_AMPA_B_dend * (E_AMPA-V_dend) * s_AMPA_B_dend * w_AMPA_B_dend  :amp
ds_AMPA_B_dend/dt = -s_AMPA_B_dend / t_AMPA_decay_B_dend  :1
I_NMDA_A_dend = g_NMDA_A_dend * (E_NMDA-V_dend) * s_NMDA_A_dend / (1 + Mg_con * exp(-
↪Alpha_NMDA*(V_dend/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_A_dend  :amp
ds_NMDA_A_dend/dt = -s_NMDA_A_dend/t_NMDA_decay_A_dend  :1
dI_noise_dend/dt = (mean_noise_dend-I_noise_dend) / tau_noise_dend + sigma_noise_dend *␣
↪(sqrt(2/(tau_noise_dend*dt)) * randn()) :amp
```

*NOTE: You can find more info about how random noise is impemented in*Brian's documentation.

2.2.5 Dendritic spikes

```
dend.dspikes('Na')
print(dend.equations)
```

```
dV_dend/dt = (gL_dend * (EL_dend-V_dend) + I_dend) / C_dend  :volt
I_dend = I_ext_dend + I_rise_Na_dend + I_fall_Na_dend + I_noise_dend + I_NMDA_A_dend + I_
↪AMPA_B_dend + I_AMPA_A_dend  :amp
I_ext_dend  :amp
I_AMPA_A_dend = g_AMPA_A_dend * (E_AMPA-V_dend) * s_AMPA_A_dend * w_AMPA_A_dend  :amp
ds_AMPA_A_dend/dt = -s_AMPA_A_dend / t_AMPA_decay_A_dend  :1
I_AMPA_B_dend = g_AMPA_B_dend * (E_AMPA-V_dend) * s_AMPA_B_dend * w_AMPA_B_dend  :amp
ds_AMPA_B_dend/dt = -s_AMPA_B_dend / t_AMPA_decay_B_dend  :1
I_NMDA_A_dend = g_NMDA_A_dend * (E_NMDA-V_dend) * s_NMDA_A_dend / (1 + Mg_con * exp(-
↪Alpha_NMDA*(V_dend/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_A_dend  :amp
ds_NMDA_A_dend/dt = -s_NMDA_A_dend/t_NMDA_decay_A_dend  :1
dI_noise_dend/dt = (mean_noise_dend-I_noise_dend) / tau_noise_dend + sigma_noise_dend *␣
↪(sqrt(2/(tau_noise_dend*dt)) * randn()) :amp
I_rise_Na_dend = g_rise_Na_dend * (E_rise_Na-V_dend)  :amp
I_fall_Na_dend = g_fall_Na_dend * (E_fall_Na-V_dend)  :amp
g_rise_Na_dend = g_rise_max_Na_dend * int(t_in_timesteps <= spiketime_Na_dend + duration_
↪rise_Na_dend) * gate_Na_dend :siemens
g_fall_Na_dend = g_fall_max_Na_dend * int(t_in_timesteps <= spiketime_Na_dend + offset_
↪fall_Na_dend + duration_fall_Na_dend) * int(t_in_timesteps >= spiketime_Na_dend +␣
↪offset_fall_Na_dend) *  gate_Na_dend :siemens
spiketime_Na_dend  :1
gate_Na_dend  :1
```

2.2.6 Connecting compartments

```
dend.connect(soma)
```

NOTE: gnore the above errors for now. They will make sense in a while.

```
print(dend.equations)
```

```
dV_dend/dt = (gL_dend * (EL_dend-V_dend) + I_dend) / C_dend  :volt
I_dend = I_ext_dend + I_soma_dend  + I_rise_Na_dend + I_fall_Na_dend + I_noise_dend + I_
↪NMDA_A_dend + I_AMPA_B_dend + I_AMPA_A_dend  :amp
I_ext_dend  :amp
I_AMPA_A_dend = g_AMPA_A_dend * (E_AMPA-V_dend) * s_AMPA_A_dend * w_AMPA_A_dend  :amp
ds_AMPA_A_dend/dt = -s_AMPA_A_dend / t_AMPA_decay_A_dend  :1
I_AMPA_B_dend = g_AMPA_B_dend * (E_AMPA-V_dend) * s_AMPA_B_dend * w_AMPA_B_dend  :amp
ds_AMPA_B_dend/dt = -s_AMPA_B_dend / t_AMPA_decay_B_dend  :1
I_NMDA_A_dend = g_NMDA_A_dend * (E_NMDA-V_dend) * s_NMDA_A_dend / (1 + Mg_con * exp(-
↪Alpha_NMDA*(V_dend/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_A_dend  :amp
ds_NMDA_A_dend/dt = -s_NMDA_A_dend/t_NMDA_decay_A_dend  :1
dI_noise_dend/dt = (mean_noise_dend-I_noise_dend) / tau_noise_dend + sigma_noise_dend *␣
↪(sqrt(2/(tau_noise_dend*dt)) * randn()) :amp
I_rise_Na_dend = g_rise_Na_dend * (E_rise_Na-V_dend)  :amp
I_fall_Na_dend = g_fall_Na_dend * (E_fall_Na-V_dend)  :amp
g_rise_Na_dend = g_rise_max_Na_dend * int(t_in_timesteps <= spiketime_Na_dend + duration_
↪rise_Na_dend) * gate_Na_dend :siemens
g_fall_Na_dend = g_fall_max_Na_dend * int(t_in_timesteps <= spiketime_Na_dend + offset_
↪fall_Na_dend + duration_fall_Na_dend) * int(t_in_timesteps >= spiketime_Na_dend +␣
↪offset_fall_Na_dend) *  gate_Na_dend :siemens
spiketime_Na_dend  :1
gate_Na_dend  :1
I_soma_dend = (V_soma-V_dend) * g_soma_dend  :amp
```

```
print(soma.equations)
```

```
dV_soma/dt = (gL_soma * (EL_soma-V_soma) + I_soma) / C_soma  :volt
I_soma = I_ext_soma + I_dend_soma  :amp
I_ext_soma  :amp
I_dend_soma = (V_dend-V_soma) * g_dend_soma  :amp
```

2.2.7 User-defined equations

Equations are Python strings, thus you can adapt them with standard string formatting practices.

```
type(dend.equations)
```

```
str
```

```
custom_model = "dcns/dt = -cns/tau_cns  :1"
eqs = f"{dend.equations}\n{custom_model}"
print(eqs)
```

```
dV_dend/dt = (gL_dend * (EL_dend-V_dend) + I_dend) / C_dend  :volt
I_dend = I_ext_dend + I_soma_dend  + I_rise_Na_dend + I_fall_Na_dend + I_noise_dend + I_
```

(continues on next page)

(continued from previous page)

```
↪NMDA_A_dend + I_AMPA_B_dend + I_AMPA_A_dend  :amp
I_ext_dend  :amp
I_AMPA_A_dend = g_AMPA_A_dend * (E_AMPA-V_dend) * s_AMPA_A_dend * w_AMPA_A_dend  :amp
ds_AMPA_A_dend/dt = -s_AMPA_A_dend / t_AMPA_decay_A_dend  :1
I_AMPA_B_dend = g_AMPA_B_dend * (E_AMPA-V_dend) * s_AMPA_B_dend * w_AMPA_B_dend  :amp
ds_AMPA_B_dend/dt = -s_AMPA_B_dend / t_AMPA_decay_B_dend  :1
I_NMDA_A_dend = g_NMDA_A_dend * (E_NMDA-V_dend) * s_NMDA_A_dend / (1 + Mg_con * exp(-
↪Alpha_NMDA*(V_dend/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_A_dend  :amp
ds_NMDA_A_dend/dt = -s_NMDA_A_dend/t_NMDA_decay_A_dend  :1
dI_noise_dend/dt = (mean_noise_dend-I_noise_dend) / tau_noise_dend + sigma_noise_dend *␣
↪(sqrt(2/(tau_noise_dend*dt)) * randn()) :amp
I_rise_Na_dend = g_rise_Na_dend * (E_rise_Na-V_dend)  :amp
I_fall_Na_dend = g_fall_Na_dend * (E_fall_Na-V_dend)  :amp
g_rise_Na_dend = g_rise_max_Na_dend * int(t_in_timesteps <= spiketime_Na_dend + duration_
↪rise_Na_dend) * gate_Na_dend :siemens
g_fall_Na_dend = g_fall_max_Na_dend * int(t_in_timesteps <= spiketime_Na_dend + offset_
↪fall_Na_dend + duration_fall_Na_dend) * int(t_in_timesteps >= spiketime_Na_dend +␣
↪offset_fall_Na_dend) *  gate_Na_dend :siemens
spiketime_Na_dend  :1
gate_Na_dend  :1
I_soma_dend = (V_soma-V_dend) * g_soma_dend  :amp
dcns/dt = -cns/tau_cns  :1
```

2.3 Setting model parameters

In this second part of the tutorial we are going to explore how to access, generate or update all model parameters.

2.3.1 Accessing model properties

```
dend.parameters
```

```
ERROR [dendrify.ephysproperties:336]
Could not calculate the g_couple for 'dend' and 'soma'.
Please make sure that [length, diameter, r_axial] are
available for both compartments.

WARNING [dendrify.ephysproperties:180]
Missing parameters [length | diameter] for 'dend'.
Could not calculate the area of 'dend', returned None.

WARNING [dendrify.ephysproperties:210]
Could not calculate the [capacitance] of 'dend', returned None.

WARNING [dendrify.ephysproperties:180]
Missing parameters [length | diameter] for 'dend'.
Could not calculate the area of 'dend', returned None.

WARNING [dendrify.ephysproperties:240]
Could not calculate the [g_leakage] of 'dend', returned None.
```

(continues on next page)

(continued from previous page)

```
ERROR [dendrify.ephysproperties:266]
Could not resolve [EL_dend] for 'dend'.

ERROR [dendrify.ephysproperties:266]
Could not resolve [C_dend] for 'dend'.

ERROR [dendrify.ephysproperties:266]
Could not resolve [gL_dend] for 'dend'.
```

```
{'w_AMPA_A_dend': 1.0,
 'w_AMPA_B_dend': 1.0,
 'w_NMDA_A_dend': 1.0,
 'tau_noise_dend': 20. * msecond,
 'sigma_noise_dend': 1. * pamp,
 'mean_noise_dend': 0. * amp,
 'g_soma_dend': None,
 'Vth_Na_dend': None,
 'g_rise_max_Na_dend': None,
 'g_fall_max_Na_dend': None,
 'E_rise_Na': None,
 'E_fall_Na': None,
 'duration_rise_Na_dend': None,
 'duration_fall_Na_dend': None,
 'offset_fall_Na_dend': None,
 'refractory_Na_dend': None,
 'E_AMPA': 0. * volt,
 'E_NMDA': 0. * volt,
 'E_GABA': -80. * mvolt,
 'E_Na': 70. * mvolt,
 'E_K': -89. * mvolt,
 'E_Ca': 136. * mvolt,
 'Mg_con': 1.0,
 'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'Gamma_NMDA': 0}
```

Dendrify is designed to fail loudly!!! Errors and warnings are raised if you try to access parameters that do not exist, or if something important is missing.

2.3.2 Default parameters

NOTE: Dendrify has a built-in library of default simulation parameters that can be views or adjusted using default_params() and update_default_params() respectively.

```
d.default_params()
```

```
{'E_AMPA': 0. * volt,
 'E_NMDA': 0. * volt,
 'E_GABA': -80. * mvolt,
 'E_Na': 70. * mvolt,
```

(continues on next page)

(continued from previous page)

```
 'E_K': -89. * mvolt,
 'E_Ca': 136. * mvolt,
 'Mg_con': 1.0,
 'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'Gamma_NMDA': 0}
```

```
d.update_default_params({"E_Ca":2023})
d.default_params()
```

```
{'E_AMPA': 0. * volt,
 'E_NMDA': 0. * volt,
 'E_GABA': -80. * mvolt,
 'E_Na': 70. * mvolt,
 'E_K': -89. * mvolt,
 'E_Ca': 2023,
 'Mg_con': 1.0,
 'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'Gamma_NMDA': 0}
```

2.3.3 Ephys parameters

In Dendrify, each compartment is treated as an open cylinder. Although an RC circuit does not have physical dimensions, length and diameter are needed to estimate a compartment's theoretical surface area.

```
soma = Soma('soma', length=20*um, diameter=20*um,
            cm=1*uF/(cm**2), gl=40*uS/(cm**2),
            r_axial=150*ohm*cm, v_rest=-70*mV)

dend = Dendrite('dend', length=20*um, diameter=20*um,
                cm=1*uF/(cm**2), gl=40*uS/(cm**2),
                r_axial=150*ohm*cm, v_rest=-70*mV)
```

```
print(soma) # no more errors :)
```

```

OBJECT
------
<class 'dendrify.compartment.Soma'>


EQUATIONS
---------
dV_soma/dt = (gL_soma * (EL_soma-V_soma) + I_soma) / C_soma  :volt
I_soma = I_ext_soma  :amp
I_ext_soma  :amp


PARAMETERS
----------
```

(continues on next page)

(continued from previous page)

```
{'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'C_soma': 12.56637061 * pfarad,
 'EL_soma': -70. * mvolt,
 'E_AMPA': 0. * volt,
 'E_Ca': 2023,
 'E_GABA': -80. * mvolt,
 'E_K': -89. * mvolt,
 'E_NMDA': 0. * volt,
 'E_Na': 70. * mvolt,
 'Gamma_NMDA': 0,
 'Mg_con': 1.0,
 'gL_soma': 0.50265482 * nsiemens}


USER PARAMETERS
---------------
{'_dimensionless': False,
 'cm': 0.01 * metre ** -4 * kilogram ** -1 * second ** 4 * amp ** 2,
 'cm_abs': None,
 'diameter': 20. * umetre,
 'gl': 0.4 * metre ** -4 * kilogram ** -1 * second ** 3 * amp ** 2,
 'gl_abs': None,
 'length': 20. * umetre,
 'name': 'soma',
 'r_axial': 1.5 * metre ** 3 * kilogram * second ** -3 * amp ** -2,
 'scale_factor': 1.0,
 'spine_factor': 1.0,
 'v_rest': -70. * mvolt}
```

```
# The surface area of an equivalent open cylinder
soma.area
```

$1256.637061435917\,\mathrm{um}^2$

```
# Absolute capacitance (specific capacitance [cm] multiplied by area)
soma.capacitance
```

$12.566370614359167\,\mathrm{pF}$

```
# Absolute leakage conductance (specific leakage conductance [gl] multiplied by area)
soma.g_leakage
```

$0.5026548245743667\,\mathrm{nS}$

2.3.4 Synaptic parameters

```
dend.synapse('AMPA', 'A', g=1*nS, t_decay=5*ms)
print(dend)
```

```
OBJECT
------
<class 'dendrify.compartment.Dendrite'>


EQUATIONS
---------
dV_dend/dt = (gL_dend * (EL_dend-V_dend) + I_dend) / C_dend  :volt
I_dend = I_ext_dend + I_AMPA_A_dend  :amp
I_ext_dend  :amp
I_AMPA_A_dend = g_AMPA_A_dend * (E_AMPA-V_dend) * s_AMPA_A_dend * w_AMPA_A_dend  :amp
ds_AMPA_A_dend/dt = -s_AMPA_A_dend / t_AMPA_decay_A_dend  :1


PARAMETERS
----------
{'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'C_dend': 12.56637061 * pfarad,
 'EL_dend': -70. * mvolt,
 'E_AMPA': 0. * volt,
 'E_Ca': 2023,
 'E_GABA': -80. * mvolt,
 'E_K': -89. * mvolt,
 'E_NMDA': 0. * volt,
 'E_Na': 70. * mvolt,
 'Gamma_NMDA': 0,
 'Mg_con': 1.0,
 'gL_dend': 0.50265482 * nsiemens,
 'g_AMPA_A_dend': 1. * nsiemens,
 't_AMPA_decay_A_dend': 5. * msecond,
 'w_AMPA_A_dend': 1.0}


EVENTS
------
[]


EVENT CONDITIONS
----------------
{}


USER PARAMETERS
---------------
{'cm': 0.01 * metre ** -4 * kilogram ** -1 * second ** 4 * amp ** 2,
```

(continues on next page)

(continued from previous page)

```
 'diameter': 20. * umetre,
 'gl': 0.4 * metre ** -4 * kilogram ** -1 * second ** 3 * amp ** 2,
 'length': 20. * umetre,
 'name': 'dend',
 'r_axial': 1.5 * metre ** 3 * kilogram * second ** -3 * amp ** -2,
 'scale_factor': 1.0,
 'spine_factor': 1.0,
 'v_rest': -70. * mvolt}
```

```
# NMDA synapse with instant activation and exponential decay:
dend.synapse('NMDA', 'A', g=1*nS, t_decay=60*ms)

# NMDA synapse as a sum of two exponentials with rise and decay kinetics:
dend.synapse('NMDA', 'B', g=1*nS, t_decay=60*ms, t_rise=5*ms)

print(dend)
```

```
OBJECT
------
<class 'dendrify.compartment.Dendrite'>


EQUATIONS
---------
dV_dend/dt = (gL_dend * (EL_dend-V_dend) + I_dend) / C_dend  :volt
I_dend = I_ext_dend + I_NMDA_B_dend + I_NMDA_A_dend + I_AMPA_A_dend  :amp
I_ext_dend  :amp
I_AMPA_A_dend = g_AMPA_A_dend * (E_AMPA-V_dend) * s_AMPA_A_dend * w_AMPA_A_dend  :amp
ds_AMPA_A_dend/dt = -s_AMPA_A_dend / t_AMPA_decay_A_dend  :1
I_NMDA_A_dend = g_NMDA_A_dend * (E_NMDA-V_dend) * s_NMDA_A_dend / (1 + Mg_con * exp(-
↪Alpha_NMDA*(V_dend/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_A_dend  :amp
ds_NMDA_A_dend/dt = -s_NMDA_A_dend/t_NMDA_decay_A_dend  :1
I_NMDA_B_dend = g_NMDA_B_dend * (E_NMDA-V_dend) * x_NMDA_B_dend / (1 + Mg_con * exp(-
↪Alpha_NMDA*(V_dend/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_B_dend  :amp
dx_NMDA_B_dend/dt = (-x_NMDA_B_dend/t_NMDA_decay_B_dend) + s_NMDA_B_dend/ms  :1
ds_NMDA_B_dend/dt = -s_NMDA_B_dend / t_NMDA_rise_B_dend  :1


PARAMETERS
----------
{'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'C_dend': 12.56637061 * pfarad,
 'EL_dend': -70. * mvolt,
 'E_AMPA': 0. * volt,
 'E_Ca': 2023,
 'E_GABA': -80. * mvolt,
 'E_K': -89. * mvolt,
 'E_NMDA': 0. * volt,
 'E_Na': 70. * mvolt,
 'Gamma_NMDA': 0,
```

(continues on next page)

(continued from previous page)

```
 'Mg_con': 1.0,
 'gL_dend': 0.50265482 * nsiemens,
 'g_AMPA_A_dend': 1. * nsiemens,
 'g_NMDA_A_dend': 1. * nsiemens,
 'g_NMDA_B_dend': 1. * nsiemens,
 't_AMPA_decay_A_dend': 5. * msecond,
 't_NMDA_decay_A_dend': 60. * msecond,
 't_NMDA_decay_B_dend': 60. * msecond,
 't_NMDA_rise_B_dend': 5. * msecond,
 'w_AMPA_A_dend': 1.0,
 'w_NMDA_A_dend': 1.0,
 'w_NMDA_B_dend': 1.0}


EVENTS
------
[]


EVENT CONDITIONS
----------------
{}


USER PARAMETERS
---------------
{'cm': 0.01 * metre ** -4 * kilogram ** -1 * second ** 4 * amp ** 2,
 'diameter': 20. * umetre,
 'gl': 0.4 * metre ** -4 * kilogram ** -1 * second ** 3 * amp ** 2,
 'length': 20. * umetre,
 'name': 'dend',
 'r_axial': 1.5 * metre ** 3 * kilogram * second ** -3 * amp ** -2,
 'scale_factor': 1.0,
 'spine_factor': 1.0,
 'v_rest': -70. * mvolt}
```

2.3.5 Random noise parameters

```
dend.noise(mean=0*pA, sigma=10*pA, tau=1*ms)
```

2.3.6 dSpike parameters

We will explore this topic in another tutorial…

2.3.7 Coupling parameters

```
# Automatic approach
soma.connect(dend, g=10*nS)
```

```
print(soma)
```

```
OBJECT
------
<class 'dendrify.compartment.Soma'>


EQUATIONS
---------
dV_soma/dt = (gL_soma * (EL_soma-V_soma) + I_soma) / C_soma  :volt
I_soma = I_ext_soma + I_dend_soma   :amp
I_ext_soma  :amp
I_dend_soma = (V_dend-V_soma) * g_dend_soma  :amp


PARAMETERS
----------
{'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'C_soma': 12.56637061 * pfarad,
 'EL_soma': -70. * mvolt,
 'E_AMPA': 0. * volt,
 'E_Ca': 2023,
 'E_GABA': -80. * mvolt,
 'E_K': -89. * mvolt,
 'E_NMDA': 0. * volt,
 'E_Na': 70. * mvolt,
 'Gamma_NMDA': 0,
 'Mg_con': 1.0,
 'gL_soma': 0.50265482 * nsiemens,
 'g_dend_soma': 10. * nsiemens}


USER PARAMETERS
---------------
{'_dimensionless': False,
 'cm': 0.01 * metre ** -4 * kilogram ** -1 * second ** 4 * amp ** 2,
 'cm_abs': None,
 'diameter': 20. * umetre,
 'gl': 0.4 * metre ** -4 * kilogram ** -1 * second ** 3 * amp ** 2,
 'gl_abs': None,
 'length': 20. * umetre,
```

(continues on next page)

(continued from previous page)

```
 'name': 'soma',
 'r_axial': 1.5 * metre ** 3 * kilogram * second ** -3 * amp ** -2,
 'scale_factor': 1.0,
 'spine_factor': 1.0,
 'v_rest': -70. * mvolt}
```

2.3.8 Dimensionless compartments

If you know the model's desired capacitance and leakage conductance you can pass them dirrectly as a parameters when creating a compartment.

```
soma = Soma('soma', cm_abs=200*pF, gl_abs=20*nS, v_rest=-70*mV)
dend = Dendrite('dend', cm_abs=200*pF, gl_abs=20*nS, v_rest=-70*mV)
```

Since these compartments have no dimensions, the automatic connection approach will not work, since it calculates the resistance between two adjucent half-cylinders.

```
soma.connect(dend)
```

```
---------------------------------------------------------------------------
DimensionlessCompartmentError             Traceback (most recent call last)
Cell In[31], line 1
----> 1 soma.connect(dend)

File ~/anaconda3/envs/dendrify/lib/python3.11/site-packages/dendrify/compartment.py:169,␣
↪in Compartment.connect(self, other, g)
    166     raise ValueError(
    167         "Cannot connect compartments with the same name.\n")
    168 if (self.dimensionless or other.dimensionless) and type(g) == str:
--> 169     raise DimensionlessCompartmentError(
    170         ("Cannot automatically calculate the coupling \nconductance of "
    171          "dimensionless compartments. To resolve this error, perform\n"
    172          "one of the following:\n\n"
    173          f"1. Provide [length, diameter, r_axial] for both '{self.name}'"
    174          f" and '{other.name}'.\n\n"
    175          f"2. Turn both compartment into dimensionless by providing only"
    176          " values for \n   [cm_abs, gl_abs] and then connect them using "
    177          "an exact coupling conductance."
    178          )
    179     )
    181 # Current from Comp2 -> Comp1
    182 I_forward = 'I_{1}_{0} = (V_{1}-V_{0}) * g_{1}_{0}  :amp'.format(
    183     self.name, other.name)

DimensionlessCompartmentError: Cannot automatically calculate the coupling
conductance of dimensionless compartments. To resolve this error, perform
one of the following:

1. Provide [length, diameter, r_axial] for both 'soma' and 'dend'.

2. Turn both compartment into dimensionless by providing only values for
   [cm_abs, gl_abs] and then connect them using an exact coupling conductance.
```

*However, you can still connect them by explicitly specifying the coupling conductance as shown bellow.***IMPORTANT: This trick also works for compartments that have dimensions as well**.

```
soma.connect(dend, g=10*nS)
```

```
print(soma)
```

```
OBJECT
------
<class 'dendrify.compartment.Soma'>


EQUATIONS
---------
dV_soma/dt = (gL_soma * (EL_soma-V_soma) + I_soma) / C_soma  :volt
I_soma = I_ext_soma + I_dend_soma   :amp
I_ext_soma  :amp
I_dend_soma = (V_dend-V_soma) * g_dend_soma  :amp


PARAMETERS
----------
{'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'C_soma': 200. * pfarad,
 'EL_soma': -70. * mvolt,
 'E_AMPA': 0. * volt,
 'E_Ca': 2023,
 'E_GABA': -80. * mvolt,
 'E_K': -89. * mvolt,
 'E_NMDA': 0. * volt,
 'E_Na': 70. * mvolt,
 'Gamma_NMDA': 0,
 'Mg_con': 1.0,
 'gL_soma': 20. * nsiemens,
 'g_dend_soma': 10. * nsiemens}


USER PARAMETERS
---------------
{'_dimensionless': True,
 'cm': None,
 'cm_abs': 200. * pfarad,
 'diameter': None,
 'gl': None,
 'gl_abs': 20. * nsiemens,
 'length': None,
 'name': 'soma',
 'r_axial': None,
 'scale_factor': None,
 'spine_factor': None,
 'v_rest': -70. * mvolt}
```

2.4 Creating point neurons

Dendrify also supports point (single-compartment) neuron models that share most of the functionalities of compartment objects (Soma/Dendrite). Notice that here, there is no need to specify a compartment's name.

```
model = PointNeuronModel(model='leakyIF', v_rest=-60*mV,
                         cm_abs=200*pF, gl_abs=10*nS)
model.noise(mean=10*pA, sigma=100*pA, tau=20*ms)
model.synapse('AMPA', tag='x', g=2*nS, t_decay=2*ms)

print(model)
```

```
OBJECT
------
<class 'dendrify.neuronmodel.PointNeuronModel'>


EQUATIONS
---------
dV/dt = (gL * (EL-V) + I) / C  :volt
I = I_ext + I_AMPA_x + I_noise  :amp
I_ext  :amp
dI_noise/dt = (mean_noise-I_noise) / tau_noise + sigma_noise * (sqrt(2/(tau_noise*dt)) *␣
↪randn()) :amp
I_AMPA_x = g_AMPA_x * (E_AMPA-V) * s_AMPA_x * w_AMPA_x  :amp
ds_AMPA_x/dt = -s_AMPA_x / t_AMPA_decay_x  :1


PARAMETERS
----------
{'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'C': 200. * pfarad,
 'EL': -60. * mvolt,
 'E_AMPA': 0. * volt,
 'E_Ca': 2023,
 'E_GABA': -80. * mvolt,
 'E_K': -89. * mvolt,
 'E_NMDA': 0. * volt,
 'E_Na': 70. * mvolt,
 'Gamma_NMDA': 0,
 'Mg_con': 1.0,
 'gL': 10. * nsiemens,
 'g_AMPA_x': 2. * nsiemens,
 'mean_noise': 10. * pamp,
 'sigma_noise': 100. * pamp,
 't_AMPA_decay_x': 2. * msecond,
 'tau_noise': 20. * msecond,
 'w_AMPA_x': 1.0}


USER PARAMETERS
```

(continues on next page)

(continued from previous page)

```
---------------
{'_dimensionless': True,
 'cm': None,
 'cm_abs': 200. * pfarad,
 'diameter': None,
 'gl': None,
 'gl_abs': 10. * nsiemens,
 'length': None,
 'name': None,
 'r_axial': None,
 'scale_factor': None,
 'spine_factor': None,
 'v_rest': -60. * mvolt}
```

CHAPTER

THREE

RUNNING SIMULATIONS

In this tutorial, we are going to cover the following topics:

- How to merge compartments into compartmental neuron models
- How to make Dendrify and Brian interact with each other
- How to run simulations of Dendrify models in Brian

Disclaimer: Below, we present a generic "toy" model developed solely for educational purposes. However, please note that its parameters and behavior have not been validated using real data. If you intend to utilize Dendrify in a project, we strongly advise against using this model as it is, unless you first calibrate its parameters to align with your specific requirements.

3.1 Imports & settings

```
import brian2 as b
import matplotlib.pyplot as plt
from brian2.units import *
from dendrify import Soma, Dendrite, NeuronModel

b.prefs.codegen.target = 'numpy' # faster for basic models and short simulations
```

```
# Plot settings
blue = '#005c94ff'
green = '#338000ff'
orange = '#ff6600ff'
notred = '#aa0044ff'
params = {
          "legend.fontsize": 10,
          "legend.handlelength": 1.5,
          "legend.edgecolor": 'inherit',
          "legend.columnspacing": 0.8,
          "legend.handletextpad": 0.5,
          "axes.labelsize": 10,
          "axes.titlesize": 11,
          "axes.spines.right": False,
          "axes.spines.top": False,
          "xtick.labelsize": 10,
          "ytick.labelsize": 10,
```

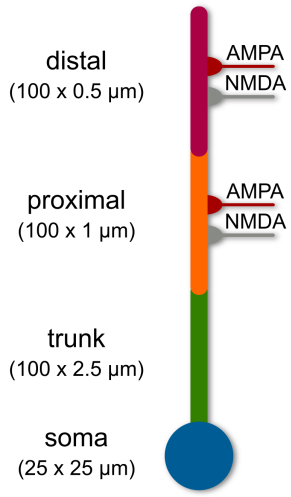
(continues on next page)

(continued from previous page)

```
          'lines.markersize': 3,
          'lines.linewidth': 1.25,
          'grid.color': "#d3d3d3",
          'figure.dpi': 150,
          'axes.prop_cycle': b.cycler(color=[blue, green, orange, notred])
          }

plt.rcParams.update(params)
```

3.2 Create a compartmental model

Lets try to recreate the following basic 4-compartment model:



According to the previous tutorial the code should look somethink like this:

```
# create soma
soma = Soma('soma', model='leakyIF', length=25*um, diameter=25*um,
            cm=1*uF/(cm**2), gl=40*uS/(cm**2),
            v_rest=-65*mV)

# create trunk
trunk = Dendrite('trunk', length=100*um, diameter=2.5*um,
                 cm=1*uF/(cm**2), gl=40*uS/(cm**2),
                 v_rest=-65*mV)

# create proximal dendrite
prox = Dendrite('prox', length=100*um, diameter=1*um,
                cm=1*uF/(cm**2), gl=40*uS/(cm**2),
                v_rest=-65*mV)
prox.synapse('AMPA', tag='pathY', g=1*nS,  t_decay=2*ms)
prox.synapse('NMDA', tag='pathY', g=1*nS,  t_decay=60*ms)

# create distal dendrite
dist = Dendrite('dist', length=100*um, diameter=0.5*um,
```

(continues on next page)

(continued from previous page)

```
                cm=1*uF/(cm**2), gl=40*uS/(cm**2),
                v_rest=-65*mV)
dist.synapse('AMPA', tag='pathX', g=1*nS,  t_decay=2*ms)
dist.synapse('NMDA', tag='pathX', g=1*nS,  t_decay=60*ms)

soma.connect(trunk, g=15*nS)
trunk.connect(prox, g=6*nS)
prox.connect(dist, g=2*nS)
```

HOWEVER: There is a far better way for creating a compartmental model!!

```
# create compartments
soma = Soma('soma', model='leakyIF', length=25*um, diameter=25*um)
trunk = Dendrite('trunk', length=100*um, diameter=2.5*um)
prox = Dendrite('prox', length=100*um, diameter=1*um)
dist = Dendrite('dist', length=100*um, diameter=0.5*um)

# add AMPA/NMDA synapses
prox.synapse('AMPA', tag='pathY', g=1*nS,  t_decay=2*ms)
prox.synapse('NMDA', tag='pathY', g=1*nS,  t_decay=60*ms)
dist.synapse('AMPA', tag='pathX', g=1*nS,  t_decay=2*ms)
dist.synapse('NMDA', tag='pathX', g=1*nS,  t_decay=60*ms)

# merge compartments into a  neuron model and set its basic properties
graph = [(soma, trunk, 15*nS), (trunk, prox, 6*nS), (prox, dist, 2*nS)]
model = NeuronModel(graph, cm=1*uF/(cm**2), gl=40*uS/(cm**2),
                    v_rest=-65*mV, scale_factor=2.8, spine_factor=1.5)
```

The NeuronModel class, not only allows to set model parameters, but also unlocks many cool functions that we are going to explore now.

```
print(model)
```

```

OBJECT
------
<class 'dendrify.neuronmodel.NeuronModel'>


EQUATIONS
---------
dV_soma/dt = (gL_soma * (EL_soma-V_soma) + I_soma) / C_soma  :volt
I_soma = I_ext_soma + I_trunk_soma   :amp
I_ext_soma  :amp
I_trunk_soma = (V_trunk-V_soma) * g_trunk_soma  :amp

dV_trunk/dt = (gL_trunk * (EL_trunk-V_trunk) + I_trunk) / C_trunk  :volt
I_trunk = I_ext_trunk + I_prox_trunk  + I_soma_trunk   :amp
I_ext_trunk  :amp
I_soma_trunk = (V_soma-V_trunk) * g_soma_trunk  :amp
I_prox_trunk = (V_prox-V_trunk) * g_prox_trunk  :amp

dV_prox/dt = (gL_prox * (EL_prox-V_prox) + I_prox) / C_prox  :volt
```

(continues on next page)

(continued from previous page)

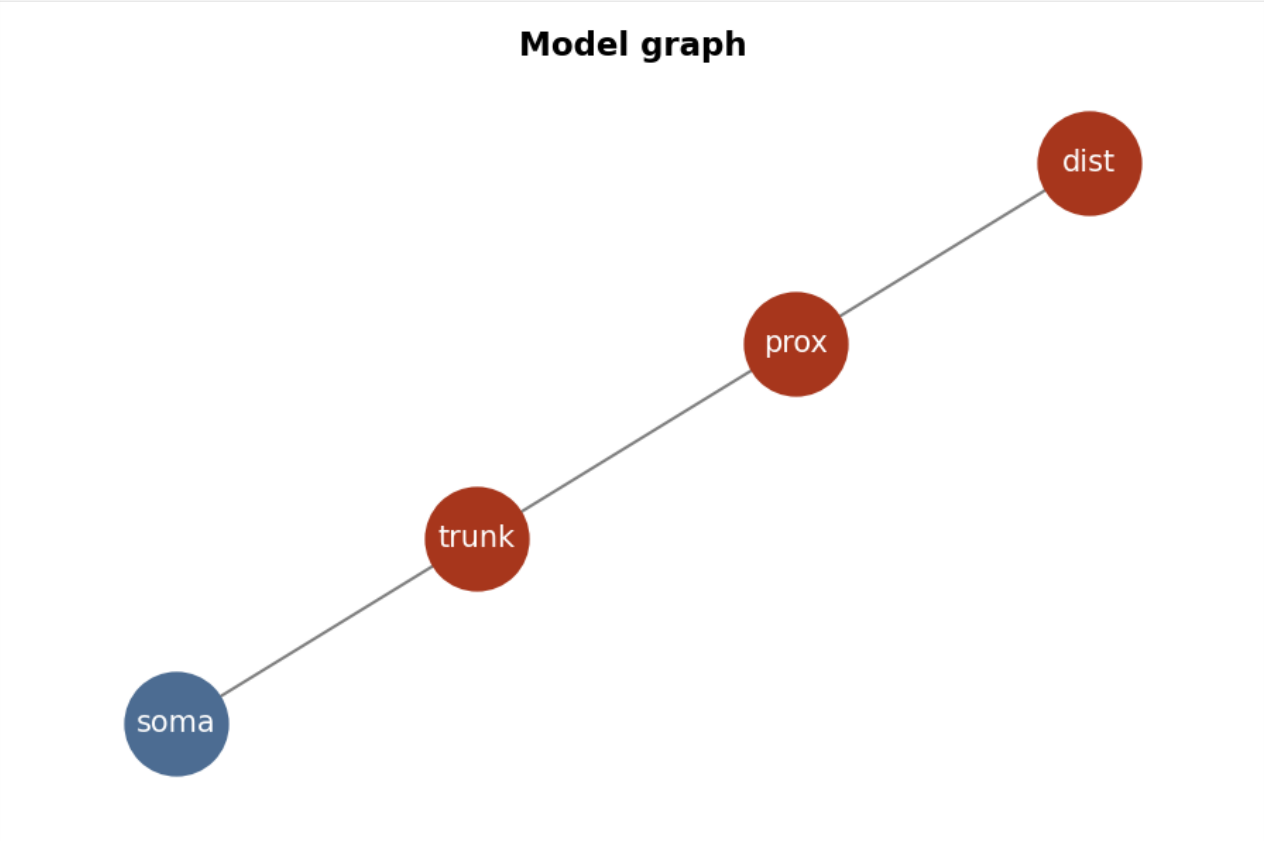
```
I_prox = I_ext_prox + I_dist_prox  + I_trunk_prox  + I_NMDA_pathY_prox + I_AMPA_pathY_
↪prox  :amp
I_ext_prox  :amp
I_AMPA_pathY_prox = g_AMPA_pathY_prox * (E_AMPA-V_prox) * s_AMPA_pathY_prox * w_AMPA_
↪pathY_prox  :amp
ds_AMPA_pathY_prox/dt = -s_AMPA_pathY_prox / t_AMPA_decay_pathY_prox  :1
I_NMDA_pathY_prox = g_NMDA_pathY_prox * (E_NMDA-V_prox) * s_NMDA_pathY_prox / (1 + Mg_
↪con * exp(-Alpha_NMDA*(V_prox/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_pathY_prox  :amp
ds_NMDA_pathY_prox/dt = -s_NMDA_pathY_prox/t_NMDA_decay_pathY_prox  :1
I_trunk_prox = (V_trunk-V_prox) * g_trunk_prox  :amp
I_dist_prox = (V_dist-V_prox) * g_dist_prox  :amp

dV_dist/dt = (gL_dist * (EL_dist-V_dist) + I_dist) / C_dist  :volt
I_dist = I_ext_dist + I_prox_dist  + I_NMDA_pathX_dist + I_AMPA_pathX_dist  :amp
I_ext_dist  :amp
I_AMPA_pathX_dist = g_AMPA_pathX_dist * (E_AMPA-V_dist) * s_AMPA_pathX_dist * w_AMPA_
↪pathX_dist  :amp
ds_AMPA_pathX_dist/dt = -s_AMPA_pathX_dist / t_AMPA_decay_pathX_dist  :1
I_NMDA_pathX_dist = g_NMDA_pathX_dist * (E_NMDA-V_dist) * s_NMDA_pathX_dist / (1 + Mg_
↪con * exp(-Alpha_NMDA*(V_dist/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_pathX_dist  :amp
ds_NMDA_pathX_dist/dt = -s_NMDA_pathX_dist/t_NMDA_decay_pathX_dist  :1
I_prox_dist = (V_prox-V_dist) * g_prox_dist  :amp


PARAMETERS
----------
{'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'C_dist': 6.59734457 * pfarad,
 'C_prox': 13.19468915 * pfarad,
 'C_soma': 82.46680716 * pfarad,
 'C_trunk': 32.98672286 * pfarad,
 'EL_dist': -65. * mvolt,
 'EL_prox': -65. * mvolt,
 'EL_soma': -65. * mvolt,
 'EL_trunk': -65. * mvolt,
 'E_AMPA': 0. * volt,
 'E_Ca': 136. * mvolt,
 'E_GABA': -80. * mvolt,
 'E_K': -89. * mvolt,
 'E_NMDA': 0. * volt,
 'E_Na': 70. * mvolt,
 'Gamma_NMDA': 0,
 'Mg_con': 1.0,
 'gL_dist': 263.8937829 * psiemens,
 'gL_prox': 0.52778757 * nsiemens,
 'gL_soma': 3.29867229 * nsiemens,
 'gL_trunk': 1.31946891 * nsiemens,
 'g_AMPA_pathX_dist': 1. * nsiemens,
 'g_AMPA_pathY_prox': 1. * nsiemens,
 'g_NMDA_pathX_dist': 1. * nsiemens,
 'g_NMDA_pathY_prox': 1. * nsiemens,
```

(continues on next page)

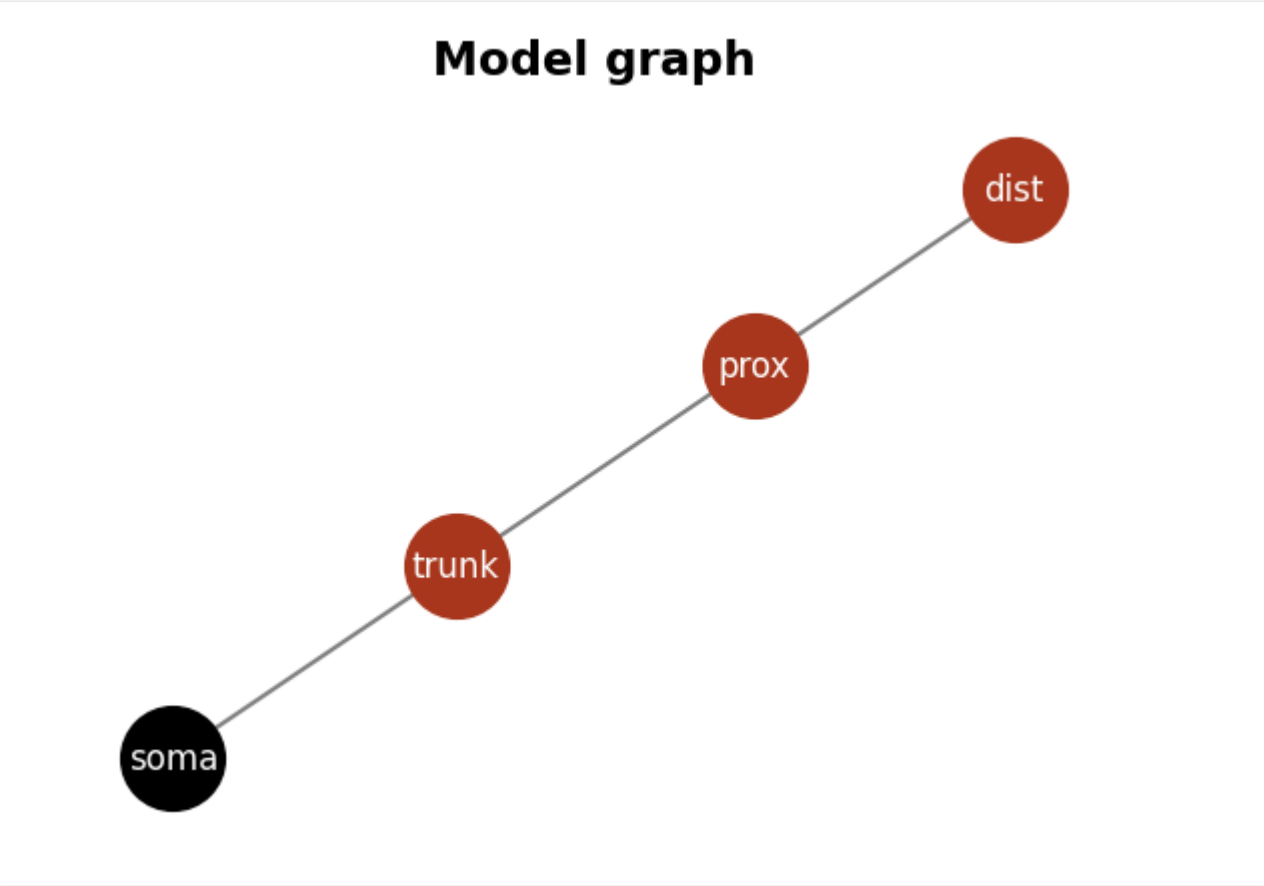
(continued from previous page)

```
 'g_dist_prox': 2. * nsiemens,
 'g_prox_dist': 2. * nsiemens,
 'g_prox_trunk': 6. * nsiemens,
 'g_soma_trunk': 15. * nsiemens,
 'g_trunk_prox': 6. * nsiemens,
 'g_trunk_soma': 15. * nsiemens,
 't_AMPA_decay_pathX_dist': 2. * msecond,
 't_AMPA_decay_pathY_prox': 2. * msecond,
 't_NMDA_decay_pathX_dist': 60. * msecond,
 't_NMDA_decay_pathY_prox': 60. * msecond,
 'w_AMPA_pathX_dist': 1.0,
 'w_AMPA_pathY_prox': 1.0,
 'w_NMDA_pathX_dist': 1.0,
 'w_NMDA_pathY_prox': 1.0}


EVENTS
------
[]


EVENT CONDITIONS
----------------
{}
```

```
model.as_graph()
```




```
model.as_graph(figsize=[4,3], scale_nodes=0.5, fontsize=8, color_soma='black')
```



3.3 Dendrify meets Brian

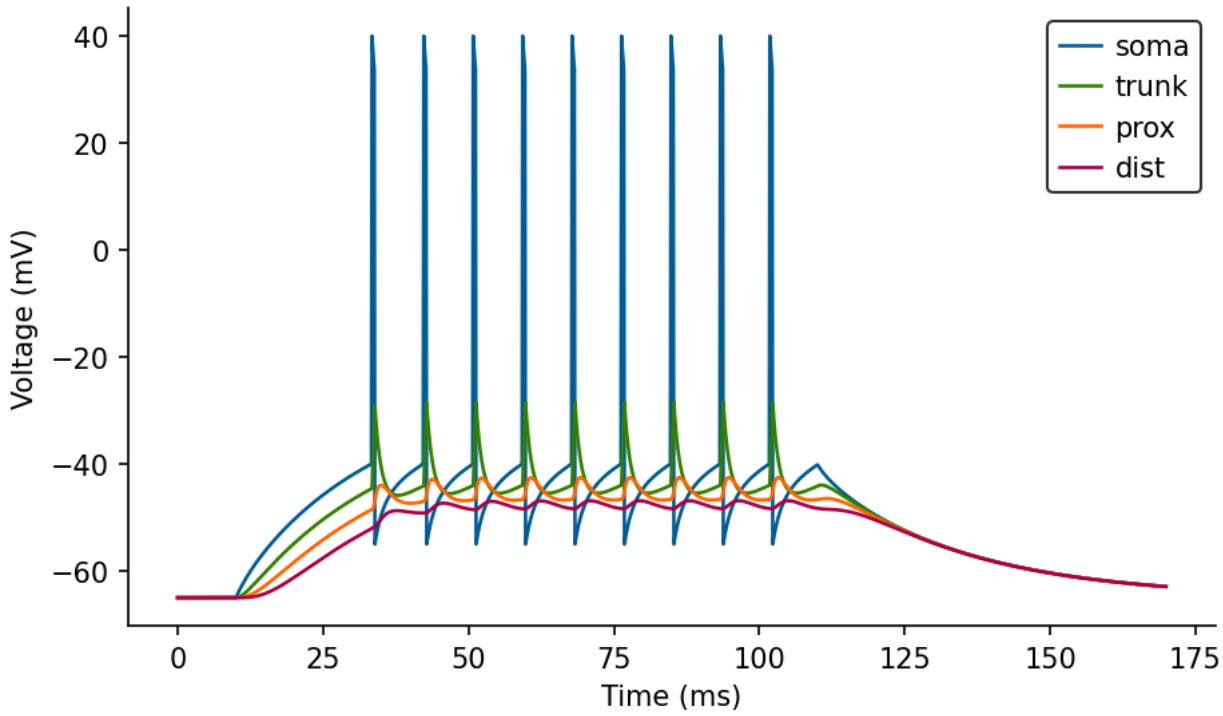
```
neuron, ap_reset = model.make_neurongroup(1, method='euler', threshold='V_soma > -40*mV',
                                  reset='V_soma = 40*mV',
                                  second_reset= 'V_soma=-55*mV',
                                  spike_width = 0.5*ms,
                                  refractory=4*ms)
```

```
# Set a monitor to record the voltages of all compartments
voltages = ['V_soma', 'V_trunk', 'V_prox', 'V_dist']
M = b.StateMonitor(neuron, voltages, record=True)
```

3.4 Run simulation and plot results

```
net = b.Network(neuron, ap_reset, M)  # organize everythink into a network
net.run(10*ms)  # no input
neuron.I_ext_soma = 200*pA
net.run(100*ms)  # 200 pA injected at the soma for 100 ms
neuron.I_ext_soma = 0*pA
net.run(60*ms) # run another 60 ms without any input
```

```
# Plot voltages
fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(M.t/ms, M.V_soma[0]/mV, label='soma')
ax.plot(M.t/ms, M.V_trunk[0]/mV, label='trunk')
ax.plot(M.t/ms, M.V_prox[0]/mV, label='prox')
ax.plot(M.t/ms, M.V_dist[0]/mV, label='dist')
ax.set_xlabel('Time (ms)')
ax.set_ylabel('Voltage (mV)')
ax.legend(loc='best');
```

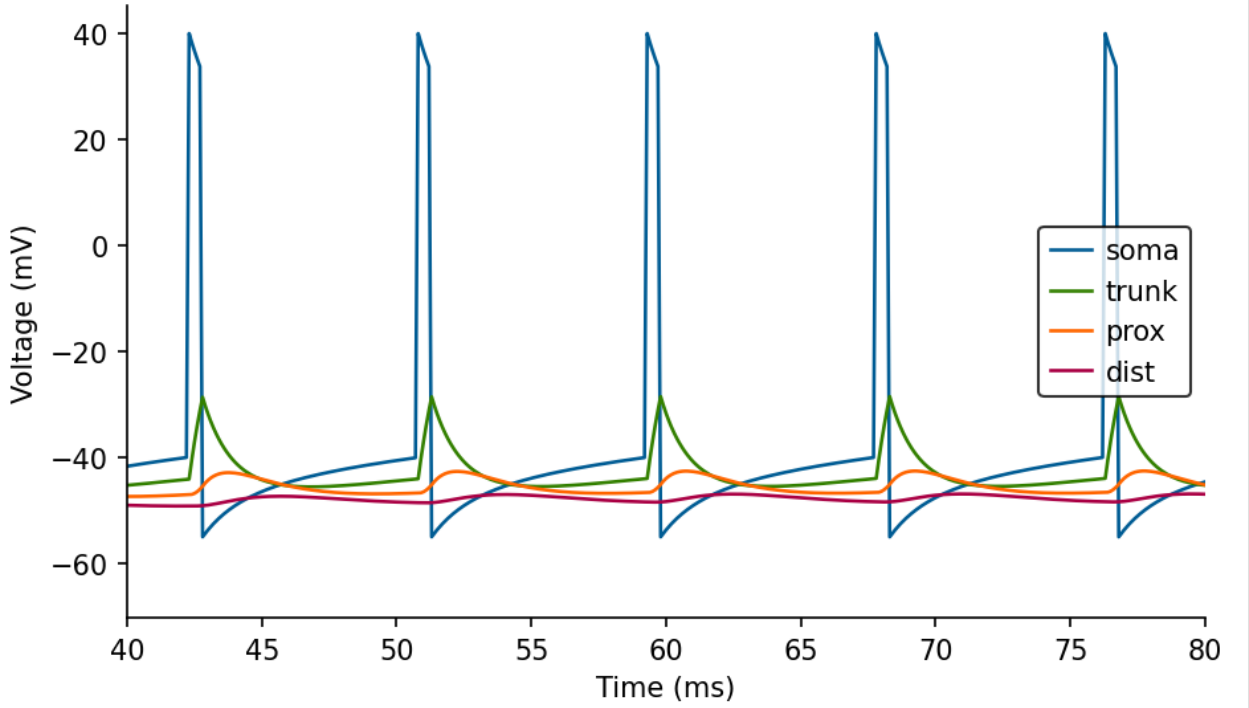



```
# Zoom in
fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(M.t/ms, M.V_soma[0]/mV, label='soma')
ax.plot(M.t/ms, M.V_trunk[0]/mV, label='trunk')
ax.plot(M.t/ms, M.V_prox[0]/mV, label='prox')
ax.plot(M.t/ms, M.V_dist[0]/mV, label='dist')
ax.set_xlabel('Time (ms)')
ax.set_ylabel('Voltage (mV)')
ax.set_xlim(left=40, right=80)
```

(continues on next page)

(continued from previous page)

```
ax.legend(loc='best');
```



NOTE: The dendritic voltage fluctuations that follow somatic APs are due to the electrical coupling of these compartments. They are not backpropagating dSpikes (not included yet in the model)

3.5 A network with random input

```
b.start_scope() # clear previous run

# First create 2 input groups
inputX = b.PoissonGroup(50, 10*Hz) # 50 neurons firing at 10 Hz
inputY = b.PoissonGroup(50, 10*Hz) # 50 neurons firing at 10 Hz

# And a NEWronGroup (I am so funny)
group, ap_reset = model.make_neurongroup(100, method='euler', threshold='V_soma > -40*mV
↪',
                                  reset='V_soma = 40*mV',
                                  second_reset= 'V_soma=-55*mV',
                                  spike_width = 0.5*ms,
                                  refractory=4*ms)
```

```
# Let's remember how the equations look like:
print(dist.equations)
```

```
dV_dist/dt = (gL_dist * (EL_dist-V_dist) + I_dist) / C_dist  :volt
I_dist = I_ext_dist + I_NMDA_pathX_dist + I_AMPA_pathX_dist  :amp
I_ext_dist  :amp
```

(continues on next page)

(continued from previous page)

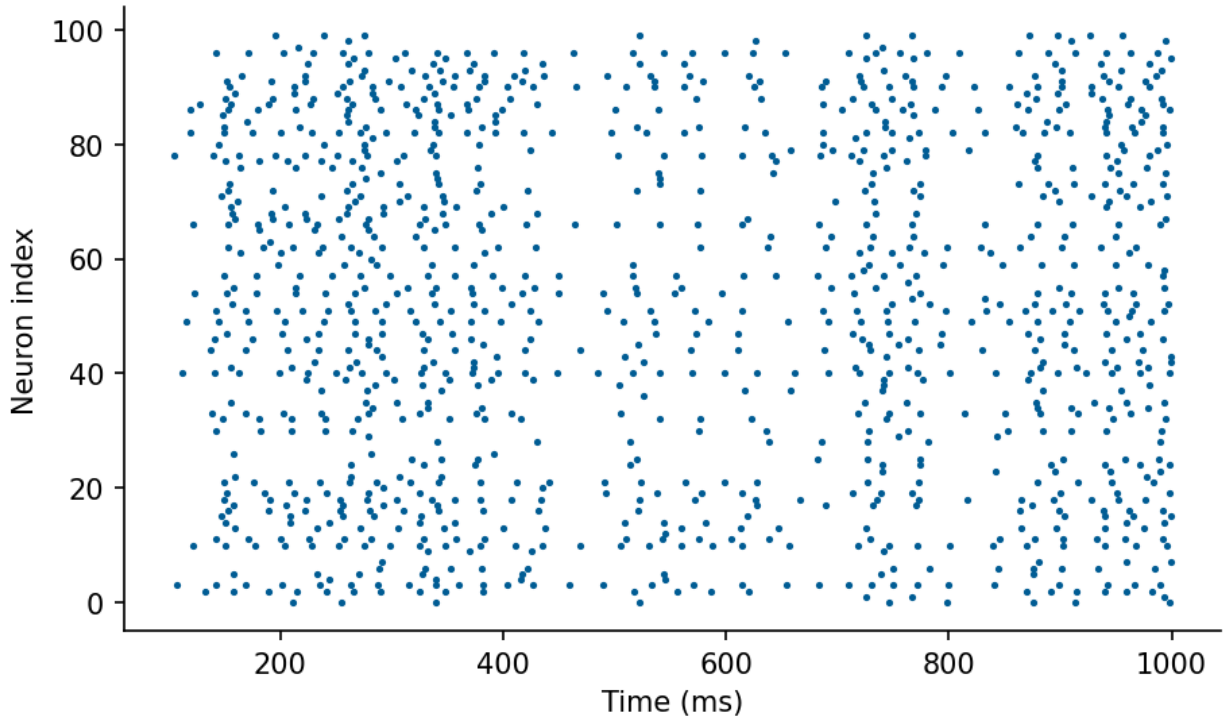
```
I_AMPA_pathX_dist = g_AMPA_pathX_dist * (E_AMPA-V_dist) * s_AMPA_pathX_dist * w_AMPA_
↪pathX_dist  :amp
ds_AMPA_pathX_dist/dt = -s_AMPA_pathX_dist / t_AMPA_decay_pathX_dist  :1
I_NMDA_pathX_dist = g_NMDA_pathX_dist * (E_NMDA-V_dist) * s_NMDA_pathX_dist / (1 + Mg_
↪con * exp(-Alpha_NMDA*(V_dist/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_pathX_dist  :amp
ds_NMDA_pathX_dist/dt = -s_NMDA_pathX_dist/t_NMDA_decay_pathX_dist  :1
```

```
# Define what happens when a presynaptic spike arrives at the synapse
synX = b.Synapses(inputX, group,
                  on_pre='s_AMPA_pathX_dist += 1; s_NMDA_pathX_dist += 1')
synY = b.Synapses(inputY, group,
                  on_pre='s_AMPA_pathY_prox += 1; s_NMDA_pathY_prox += 1')

# This is the actual connection part
synX.connect(p=0.5) # 50% of the inputs X connect to the group
synY.connect(p=0.5) # 50% of the inputs Y connect to the group
```

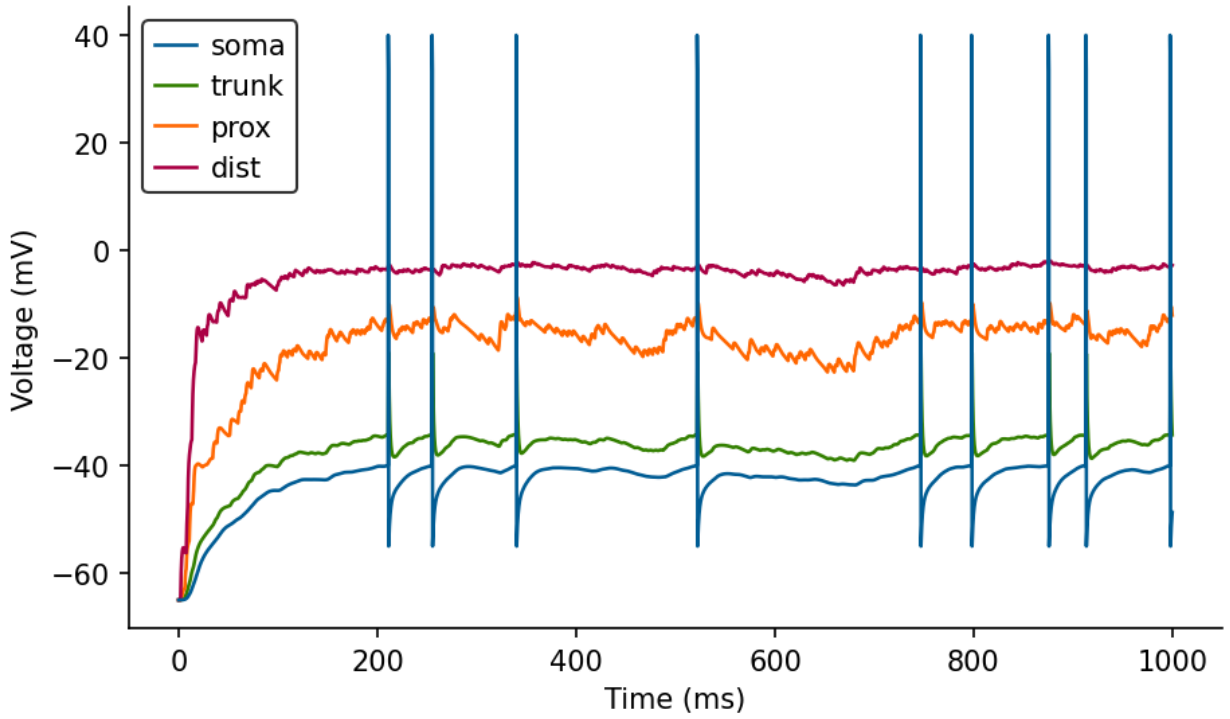
```
# Set spike and voltage monitors
S = b.SpikeMonitor(group)

voltages = ['V_soma', 'V_trunk', 'V_prox', 'V_dist']
M = b.StateMonitor(group, voltages, record=True)
```

```
# Run simulation
net = b.Network(group, ap_reset, synX, synY, S)
b.run(1000*ms)
```

```
# Plot spiketimes
fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(S.t/ms, S.i, '.')
ax.set_xlabel('Time (ms)')
ax.set_ylabel('Neuron index');
```




```
# Plot voltages
fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(M.t/ms, M.V_soma[0]/mV, label='soma', zorder=3)
ax.plot(M.t/ms, M.V_trunk[0]/mV, label='trunk')
ax.plot(M.t/ms, M.V_prox[0]/mV, label='prox')
ax.plot(M.t/ms, M.V_dist[0]/mV, label='dist')
ax.set_xlabel('Time (ms)')
ax.set_ylabel('Voltage (mV)')
ax.legend(loc='best');
```



3.6 Playing with dSpikes

```
b.start_scope() # clear previous run

# add channels to compartments
dist.dspikes('Na', g_rise=3.7*nS, g_fall=2.4*nS)
prox.dspikes('Na', g_rise=9*nS, g_fall=5.7*nS)
trunk.dspikes('Na', g_rise=22*nS, g_fall=14*nS)


model = NeuronModel(graph, cm=1*uF/(cm**2), gl=40*uS/(cm**2),
                    v_rest=-65*mV, r_axial=150*ohm*cm,
                    scale_factor=2.8, spine_factor=1.5)

model.config_dspikes('Na', threshold=-35*mV,
                     duration_rise=1.2*ms, duration_fall=2.4*ms,
                     offset_fall=0.2*ms, refractory=5*ms,
                     reversal_rise='E_Na', reversal_fall='E_K')
```

```
print(model)
```

```
OBJECT
------
<class 'dendrify.neuronmodel.NeuronModel'>
```

(continues on next page)

(continued from previous page)

```
EQUATIONS
---------
dV_soma/dt = (gL_soma * (EL_soma-V_soma) + I_soma) / C_soma  :volt
I_soma = I_ext_soma + I_trunk_soma   :amp
I_ext_soma  :amp
I_trunk_soma = (V_trunk-V_soma) * g_trunk_soma  :amp

dV_trunk/dt = (gL_trunk * (EL_trunk-V_trunk) + I_trunk) / C_trunk  :volt
I_trunk = I_ext_trunk + I_prox_trunk  + I_soma_trunk  + I_rise_Na_trunk + I_fall_Na_
↪trunk  :amp
I_ext_trunk  :amp
I_rise_Na_trunk = g_rise_Na_trunk * (E_rise_Na-V_trunk)  :amp
I_fall_Na_trunk = g_fall_Na_trunk * (E_fall_Na-V_trunk)  :amp
g_rise_Na_trunk = g_rise_max_Na_trunk * int(t_in_timesteps <= spiketime_Na_trunk +␣
↪duration_rise_Na_trunk) * gate_Na_trunk :siemens
g_fall_Na_trunk = g_fall_max_Na_trunk * int(t_in_timesteps <= spiketime_Na_trunk +␣
↪offset_fall_Na_trunk + duration_fall_Na_trunk) * int(t_in_timesteps >= spiketime_Na_
↪trunk + offset_fall_Na_trunk) *  gate_Na_trunk :siemens
spiketime_Na_trunk  :1
gate_Na_trunk  :1
I_soma_trunk = (V_soma-V_trunk) * g_soma_trunk  :amp
I_prox_trunk = (V_prox-V_trunk) * g_prox_trunk  :amp

dV_prox/dt = (gL_prox * (EL_prox-V_prox) + I_prox) / C_prox  :volt
I_prox = I_ext_prox + I_dist_prox  + I_trunk_prox  + I_rise_Na_prox + I_fall_Na_prox + I_
↪NMDA_pathY_prox + I_AMPA_pathY_prox  :amp
I_ext_prox  :amp
I_AMPA_pathY_prox = g_AMPA_pathY_prox * (E_AMPA-V_prox) * s_AMPA_pathY_prox * w_AMPA_
↪pathY_prox  :amp
ds_AMPA_pathY_prox/dt = -s_AMPA_pathY_prox / t_AMPA_decay_pathY_prox  :1
I_NMDA_pathY_prox = g_NMDA_pathY_prox * (E_NMDA-V_prox) * s_NMDA_pathY_prox / (1 + Mg_
↪con * exp(-Alpha_NMDA*(V_prox/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_pathY_prox  :amp
ds_NMDA_pathY_prox/dt = -s_NMDA_pathY_prox/t_NMDA_decay_pathY_prox  :1
I_rise_Na_prox = g_rise_Na_prox * (E_rise_Na-V_prox)  :amp
I_fall_Na_prox = g_fall_Na_prox * (E_fall_Na-V_prox)  :amp
g_rise_Na_prox = g_rise_max_Na_prox * int(t_in_timesteps <= spiketime_Na_prox + duration_
↪rise_Na_prox) * gate_Na_prox :siemens
g_fall_Na_prox = g_fall_max_Na_prox * int(t_in_timesteps <= spiketime_Na_prox + offset_
↪fall_Na_prox + duration_fall_Na_prox) * int(t_in_timesteps >= spiketime_Na_prox +␣
↪offset_fall_Na_prox) *  gate_Na_prox :siemens
spiketime_Na_prox  :1
gate_Na_prox  :1
I_trunk_prox = (V_trunk-V_prox) * g_trunk_prox  :amp
I_dist_prox = (V_dist-V_prox) * g_dist_prox  :amp

dV_dist/dt = (gL_dist * (EL_dist-V_dist) + I_dist) / C_dist  :volt
I_dist = I_ext_dist + I_prox_dist  + I_rise_Na_dist + I_fall_Na_dist + I_NMDA_pathX_dist␣
↪+ I_AMPA_pathX_dist  :amp
I_ext_dist  :amp
I_AMPA_pathX_dist = g_AMPA_pathX_dist * (E_AMPA-V_dist) * s_AMPA_pathX_dist * w_AMPA_
↪pathX_dist  :amp
```

(continues on next page)

(continued from previous page)

```
ds_AMPA_pathX_dist/dt = -s_AMPA_pathX_dist / t_AMPA_decay_pathX_dist  :1
I_NMDA_pathX_dist = g_NMDA_pathX_dist * (E_NMDA-V_dist) * s_NMDA_pathX_dist / (1 + Mg_
↪con * exp(-Alpha_NMDA*(V_dist/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_pathX_dist  :amp
ds_NMDA_pathX_dist/dt = -s_NMDA_pathX_dist/t_NMDA_decay_pathX_dist  :1
I_rise_Na_dist = g_rise_Na_dist * (E_rise_Na-V_dist)  :amp
I_fall_Na_dist = g_fall_Na_dist * (E_fall_Na-V_dist)  :amp
g_rise_Na_dist = g_rise_max_Na_dist * int(t_in_timesteps <= spiketime_Na_dist + duration_
↪rise_Na_dist) * gate_Na_dist :siemens
g_fall_Na_dist = g_fall_max_Na_dist * int(t_in_timesteps <= spiketime_Na_dist + offset_
↪fall_Na_dist + duration_fall_Na_dist) * int(t_in_timesteps >= spiketime_Na_dist +␣
↪offset_fall_Na_dist) *  gate_Na_dist :siemens
spiketime_Na_dist  :1
gate_Na_dist  :1
I_prox_dist = (V_prox-V_dist) * g_prox_dist  :amp


PARAMETERS
----------
{'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
 'C_dist': 6.59734457 * pfarad,
 'C_prox': 13.19468915 * pfarad,
 'C_soma': 82.46680716 * pfarad,
 'C_trunk': 32.98672286 * pfarad,
 'EL_dist': -65. * mvolt,
 'EL_prox': -65. * mvolt,
 'EL_soma': -65. * mvolt,
 'EL_trunk': -65. * mvolt,
 'E_AMPA': 0. * volt,
 'E_Ca': 136. * mvolt,
 'E_GABA': -80. * mvolt,
 'E_K': -89. * mvolt,
 'E_NMDA': 0. * volt,
 'E_Na': 70. * mvolt,
 'E_fall_Na': -89. * mvolt,
 'E_rise_Na': 70. * mvolt,
 'Gamma_NMDA': 0,
 'Mg_con': 1.0,
 'Vth_Na_dist': -35. * mvolt,
 'Vth_Na_prox': -35. * mvolt,
 'Vth_Na_trunk': -35. * mvolt,
 'duration_fall_Na_dist': 24,
 'duration_fall_Na_prox': 24,
 'duration_fall_Na_trunk': 24,
 'duration_rise_Na_dist': 12,
 'duration_rise_Na_prox': 12,
 'duration_rise_Na_trunk': 12,
 'gL_dist': 263.8937829 * psiemens,
 'gL_prox': 0.52778757 * nsiemens,
 'gL_soma': 3.29867229 * nsiemens,
 'gL_trunk': 1.31946891 * nsiemens,
 'g_AMPA_pathX_dist': 1. * nsiemens,
```

(continues on next page)

(continued from previous page)

```
 'g_AMPA_pathY_prox': 1. * nsiemens,
 'g_NMDA_pathX_dist': 1. * nsiemens,
 'g_NMDA_pathY_prox': 1. * nsiemens,
 'g_dist_prox': 2. * nsiemens,
 'g_fall_max_Na_dist': 2.4 * nsiemens,
 'g_fall_max_Na_prox': 5.7 * nsiemens,
 'g_fall_max_Na_trunk': 14. * nsiemens,
 'g_prox_dist': 2. * nsiemens,
 'g_prox_trunk': 6. * nsiemens,
 'g_rise_max_Na_dist': 3.7 * nsiemens,
 'g_rise_max_Na_prox': 9. * nsiemens,
 'g_rise_max_Na_trunk': 22. * nsiemens,
 'g_soma_trunk': 15. * nsiemens,
 'g_trunk_prox': 6. * nsiemens,
 'g_trunk_soma': 15. * nsiemens,
 'offset_fall_Na_dist': 2,
 'offset_fall_Na_prox': 2,
 'offset_fall_Na_trunk': 2,
 'refractory_Na_dist': 50,
 'refractory_Na_prox': 50,
 'refractory_Na_trunk': 50,
 't_AMPA_decay_pathX_dist': 2. * msecond,
 't_AMPA_decay_pathY_prox': 2. * msecond,
 't_NMDA_decay_pathX_dist': 60. * msecond,
 't_NMDA_decay_pathY_prox': 60. * msecond,
 'w_AMPA_pathX_dist': 1.0,
 'w_AMPA_pathY_prox': 1.0,
 'w_NMDA_pathX_dist': 1.0,
 'w_NMDA_pathY_prox': 1.0}


EVENTS
------
['spike_Na_trunk', 'spike_Na_prox', 'spike_Na_dist']


EVENT CONDITIONS
----------------
{'spike_Na_dist': 'V_dist >= Vth_Na_dist and t_in_timesteps >= spiketime_Na_dist +␣
↪refractory_Na_dist * gate_Na_dist',
 'spike_Na_prox': 'V_prox >= Vth_Na_prox and t_in_timesteps >= spiketime_Na_prox +␣
↪refractory_Na_prox * gate_Na_prox',
 'spike_Na_trunk': 'V_trunk >= Vth_Na_trunk and t_in_timesteps >= spiketime_Na_trunk +␣
↪refractory_Na_trunk * '
                   'gate_Na_trunk'}
```

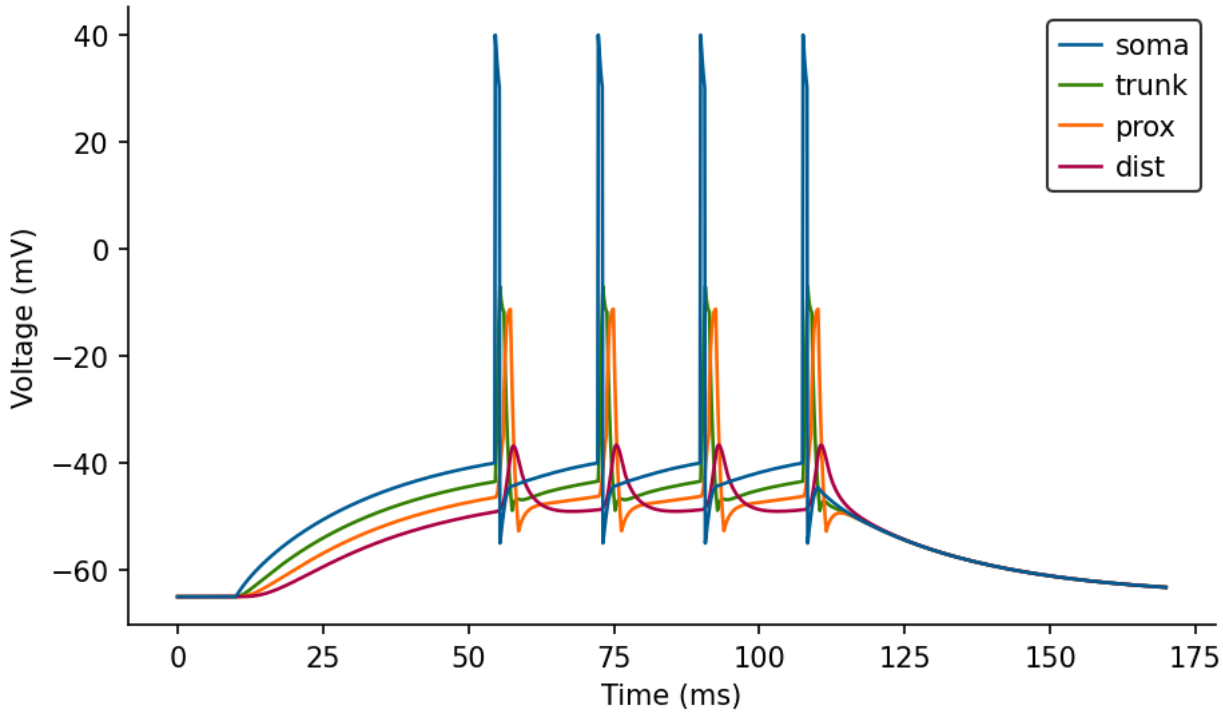
```
# Make a new neurongroup
neuron, ap_reset = model.make_neurongroup(1, method='euler', threshold='V_soma > -40*mV',
                                  reset='V_soma = 40*mV',
```

(continues on next page)

(continued from previous page)

```
                                second_reset= 'V_soma=-55*mV',
                                spike_width = 0.8*ms,
                                refractory=4*ms)

vars = ['V_soma', 'V_trunk', 'V_prox', 'V_dist']
M = b.StateMonitor(neuron, vars, record=True)

net = b.Network(neuron, ap_reset, M)
net.run(10*ms)
neuron.I_ext_soma = 150*pA
net.run(100*ms)
neuron.I_ext_soma = 0*pA
net.run(60*ms)
```

```
# @title Plot voltages
fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(M.t/ms, M.V_soma[0]/mV, label='soma', zorder=3)
ax.plot(M.t/ms, M.V_trunk[0]/mV, label='trunk')
ax.plot(M.t/ms, M.V_prox[0]/mV, label='prox')
ax.plot(M.t/ms, M.V_dist[0]/mV, label='dist')
ax.set_xlabel('Time (ms)')
ax.set_ylabel('Voltage (mV)')
ax.legend(loc='best');
```



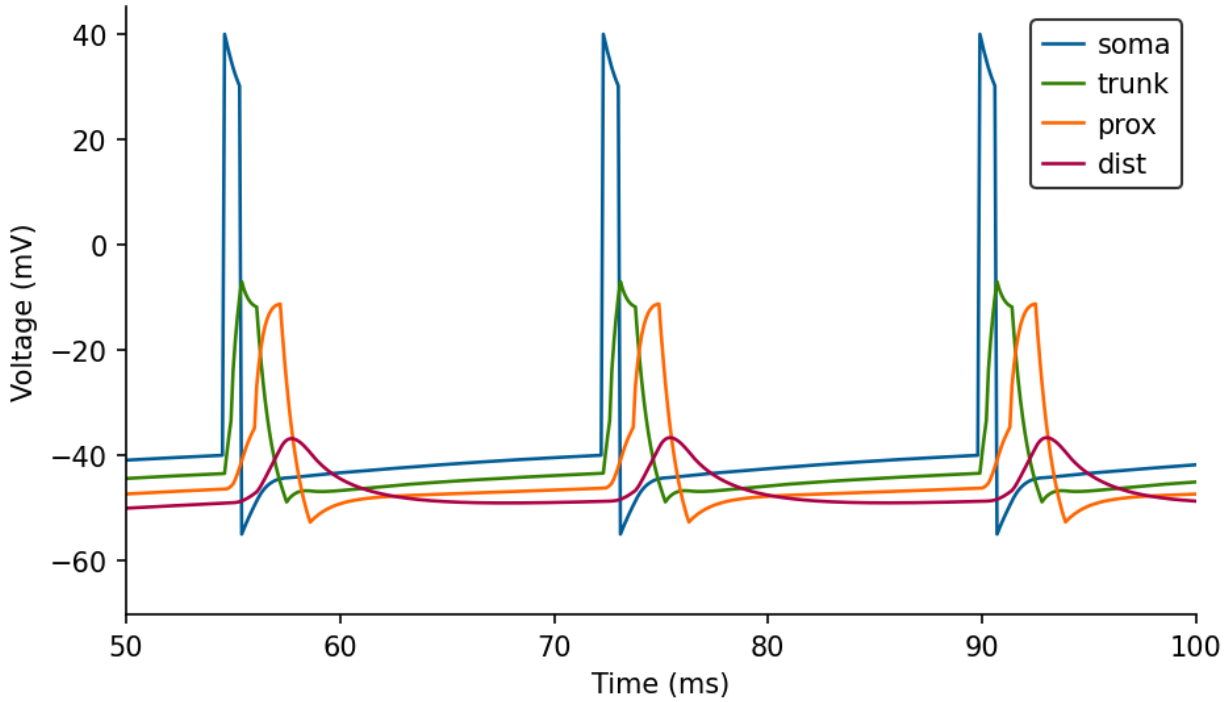
Now these are some actual backpropagating dendritic spikes with sodium-like characteristics.

```
# Zoom in
fig, ax = plt.subplots(figsize=(7, 4))
```

(continues on next page)

(continued from previous page)

```
ax.plot(M.t/ms, M.V_soma[0]/mV, label='soma')
ax.plot(M.t/ms, M.V_trunk[0]/mV, label='trunk')
ax.plot(M.t/ms, M.V_prox[0]/mV, label='prox')
ax.plot(M.t/ms, M.V_dist[0]/mV, label='dist')
ax.set_xlabel('Time (ms)')
ax.set_ylabel('Voltage (mV)')
ax.set_xlim(left=50, right=100)
ax.legend(loc='best');
```



3.7 Adding some noise

```
b.start_scope() # clear previous run

# add noise
a = 2
soma.noise(mean=a*25*pA, sigma=25*pA, tau=1*ms)
trunk.noise(mean=a*20*pA, sigma=20*pA, tau=1*ms)
prox.noise(mean=a*15*pA, sigma=15*pA, tau=1*ms)
dist.noise(mean=a*6*pA, sigma=10*pA, tau=1*ms)


# merge compartments into a  neuron model and set its basic properties
edges = [(soma, trunk, 15*nS), (trunk, prox, 8*nS), (prox, dist, 3*nS)]
model = NeuronModel(edges, cm=1*uF/(cm**2), gl=40*uS/(cm**2),
                    v_rest=-65*mV, r_axial=150*ohm*cm,
                    scale_factor=2.8, spine_factor=1.5)
```

(continues on next page)

(continued from previous page)

```
model.config_dspikes('Na', threshold=-35*mV,
                     duration_rise=1.2*ms, duration_fall=2.4*ms,
                     offset_fall=0.2*ms, refractory=5*ms,
                     reversal_rise='E_Na', reversal_fall='E_K')

# make a neuron group
neuron, ap_reset = model.make_neurongroup(1, method='euler', threshold='V_soma > -40*mV',
                                reset='V_soma = 40*mV',
                                second_reset= 'V_soma=-55*mV',
                                spike_width = 0.8*ms,
                                refractory=4*ms)

# record voltages
vars = ['V_soma', 'V_trunk', 'V_prox', 'V_dist']
M = b.StateMonitor(neuron, vars, record=True)
```

```
print(model)
```

```

OBJECT
------
<class 'dendrify.neuronmodel.NeuronModel'>


EQUATIONS
---------
dV_soma/dt = (gL_soma * (EL_soma-V_soma) + I_soma) / C_soma  :volt
I_soma = I_ext_soma + I_trunk_soma  + I_noise_soma  :amp
I_ext_soma  :amp
dI_noise_soma/dt = (mean_noise_soma-I_noise_soma) / tau_noise_soma + sigma_noise_soma *␣
↪(sqrt(2/(tau_noise_soma*dt)) * randn()) :amp
I_trunk_soma = (V_trunk-V_soma) * g_trunk_soma  :amp

dV_trunk/dt = (gL_trunk * (EL_trunk-V_trunk) + I_trunk) / C_trunk  :volt
I_trunk = I_ext_trunk + I_prox_trunk  + I_soma_trunk  + I_noise_trunk + I_rise_Na_trunk␣
↪+ I_fall_Na_trunk  :amp
I_ext_trunk  :amp
I_rise_Na_trunk = g_rise_Na_trunk * (E_rise_Na-V_trunk)  :amp
I_fall_Na_trunk = g_fall_Na_trunk * (E_fall_Na-V_trunk)  :amp
g_rise_Na_trunk = g_rise_max_Na_trunk * int(t_in_timesteps <= spiketime_Na_trunk +␣
↪duration_rise_Na_trunk) * gate_Na_trunk :siemens
g_fall_Na_trunk = g_fall_max_Na_trunk * int(t_in_timesteps <= spiketime_Na_trunk +␣
↪offset_fall_Na_trunk + duration_fall_Na_trunk) * int(t_in_timesteps >= spiketime_Na_
↪trunk + offset_fall_Na_trunk) *  gate_Na_trunk :siemens
spiketime_Na_trunk  :1
gate_Na_trunk  :1
dI_noise_trunk/dt = (mean_noise_trunk-I_noise_trunk) / tau_noise_trunk + sigma_noise_
↪trunk * (sqrt(2/(tau_noise_trunk*dt)) * randn()) :amp
I_soma_trunk = (V_soma-V_trunk) * g_soma_trunk  :amp
I_prox_trunk = (V_prox-V_trunk) * g_prox_trunk  :amp
```

(continues on next page)

(continued from previous page)

```
dV_prox/dt = (gL_prox * (EL_prox-V_prox) + I_prox) / C_prox  :volt
I_prox = I_ext_prox + I_dist_prox  + I_trunk_prox  + I_noise_prox + I_rise_Na_prox + I_
↪fall_Na_prox + I_NMDA_pathY_prox + I_AMPA_pathY_prox  :amp
I_ext_prox  :amp
I_AMPA_pathY_prox = g_AMPA_pathY_prox * (E_AMPA-V_prox) * s_AMPA_pathY_prox * w_AMPA_
↪pathY_prox  :amp
ds_AMPA_pathY_prox/dt = -s_AMPA_pathY_prox / t_AMPA_decay_pathY_prox  :1
I_NMDA_pathY_prox = g_NMDA_pathY_prox * (E_NMDA-V_prox) * s_NMDA_pathY_prox / (1 + Mg_
↪con * exp(-Alpha_NMDA*(V_prox/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_pathY_prox  :amp
ds_NMDA_pathY_prox/dt = -s_NMDA_pathY_prox/t_NMDA_decay_pathY_prox  :1
I_rise_Na_prox = g_rise_Na_prox * (E_rise_Na-V_prox)  :amp
I_fall_Na_prox = g_fall_Na_prox * (E_fall_Na-V_prox)  :amp
g_rise_Na_prox = g_rise_max_Na_prox * int(t_in_timesteps <= spiketime_Na_prox + duration_
↪rise_Na_prox) * gate_Na_prox :siemens
g_fall_Na_prox = g_fall_max_Na_prox * int(t_in_timesteps <= spiketime_Na_prox + offset_
↪fall_Na_prox + duration_fall_Na_prox) * int(t_in_timesteps >= spiketime_Na_prox +␣
↪offset_fall_Na_prox) *  gate_Na_prox :siemens
spiketime_Na_prox  :1
gate_Na_prox  :1
dI_noise_prox/dt = (mean_noise_prox-I_noise_prox) / tau_noise_prox + sigma_noise_prox *␣
↪(sqrt(2/(tau_noise_prox*dt)) * randn()) :amp
I_trunk_prox = (V_trunk-V_prox) * g_trunk_prox  :amp
I_dist_prox = (V_dist-V_prox) * g_dist_prox  :amp

dV_dist/dt = (gL_dist * (EL_dist-V_dist) + I_dist) / C_dist  :volt
I_dist = I_ext_dist + I_prox_dist  + I_noise_dist + I_rise_Na_dist + I_fall_Na_dist + I_
↪NMDA_pathX_dist + I_AMPA_pathX_dist  :amp
I_ext_dist  :amp
I_AMPA_pathX_dist = g_AMPA_pathX_dist * (E_AMPA-V_dist) * s_AMPA_pathX_dist * w_AMPA_
↪pathX_dist  :amp
ds_AMPA_pathX_dist/dt = -s_AMPA_pathX_dist / t_AMPA_decay_pathX_dist  :1
I_NMDA_pathX_dist = g_NMDA_pathX_dist * (E_NMDA-V_dist) * s_NMDA_pathX_dist / (1 + Mg_
↪con * exp(-Alpha_NMDA*(V_dist/mV+Gamma_NMDA)) / Beta_NMDA) * w_NMDA_pathX_dist  :amp
ds_NMDA_pathX_dist/dt = -s_NMDA_pathX_dist/t_NMDA_decay_pathX_dist  :1
I_rise_Na_dist = g_rise_Na_dist * (E_rise_Na-V_dist)  :amp
I_fall_Na_dist = g_fall_Na_dist * (E_fall_Na-V_dist)  :amp
g_rise_Na_dist = g_rise_max_Na_dist * int(t_in_timesteps <= spiketime_Na_dist + duration_
↪rise_Na_dist) * gate_Na_dist :siemens
g_fall_Na_dist = g_fall_max_Na_dist * int(t_in_timesteps <= spiketime_Na_dist + offset_
↪fall_Na_dist + duration_fall_Na_dist) * int(t_in_timesteps >= spiketime_Na_dist +␣
↪offset_fall_Na_dist) *  gate_Na_dist :siemens
spiketime_Na_dist  :1
gate_Na_dist  :1
dI_noise_dist/dt = (mean_noise_dist-I_noise_dist) / tau_noise_dist + sigma_noise_dist *␣
↪(sqrt(2/(tau_noise_dist*dt)) * randn()) :amp
I_prox_dist = (V_prox-V_dist) * g_prox_dist  :amp


PARAMETERS
----------
{'Alpha_NMDA': 0.062,
 'Beta_NMDA': 3.57,
```

(continues on next page)

(continued from previous page)

```
'C_dist': 6.59734457 * pfarad,
'C_prox': 13.19468915 * pfarad,
'C_soma': 82.46680716 * pfarad,
'C_trunk': 32.98672286 * pfarad,
'EL_dist': -65. * mvolt,
'EL_prox': -65. * mvolt,
'EL_soma': -65. * mvolt,
'EL_trunk': -65. * mvolt,
'E_AMPA': 0. * volt,
'E_Ca': 136. * mvolt,
'E_GABA': -80. * mvolt,
'E_K': -89. * mvolt,
'E_NMDA': 0. * volt,
'E_Na': 70. * mvolt,
'E_fall_Na': -89. * mvolt,
'E_rise_Na': 70. * mvolt,
'Gamma_NMDA': 0,
'Mg_con': 1.0,
'Vth_Na_dist': -35. * mvolt,
'Vth_Na_prox': -35. * mvolt,
'Vth_Na_trunk': -35. * mvolt,
'duration_fall_Na_dist': 24,
'duration_fall_Na_prox': 24,
'duration_fall_Na_trunk': 24,
'duration_rise_Na_dist': 12,
'duration_rise_Na_prox': 12,
'duration_rise_Na_trunk': 12,
'gL_dist': 263.8937829 * psiemens,
'gL_prox': 0.52778757 * nsiemens,
'gL_soma': 3.29867229 * nsiemens,
'gL_trunk': 1.31946891 * nsiemens,
'g_AMPA_pathX_dist': 1. * nsiemens,
'g_AMPA_pathY_prox': 1. * nsiemens,
'g_NMDA_pathX_dist': 1. * nsiemens,
'g_NMDA_pathY_prox': 1. * nsiemens,
'g_dist_prox': 3. * nsiemens,
'g_fall_max_Na_dist': 2.4 * nsiemens,
'g_fall_max_Na_prox': 5.7 * nsiemens,
'g_fall_max_Na_trunk': 14. * nsiemens,
'g_prox_dist': 3. * nsiemens,
'g_prox_trunk': 8. * nsiemens,
'g_rise_max_Na_dist': 3.7 * nsiemens,
'g_rise_max_Na_prox': 9. * nsiemens,
'g_rise_max_Na_trunk': 22. * nsiemens,
'g_soma_trunk': 15. * nsiemens,
'g_trunk_prox': 8. * nsiemens,
'g_trunk_soma': 15. * nsiemens,
'mean_noise_dist': 12. * pamp,
'mean_noise_prox': 30. * pamp,
'mean_noise_soma': 50. * pamp,
'mean_noise_trunk': 40. * pamp,
'offset_fall_Na_dist': 2,
```

(continues on next page)

(continued from previous page)

```
 'offset_fall_Na_prox': 2,
 'offset_fall_Na_trunk': 2,
 'refractory_Na_dist': 50,
 'refractory_Na_prox': 50,
 'refractory_Na_trunk': 50,
 'sigma_noise_dist': 10. * pamp,
 'sigma_noise_prox': 15. * pamp,
 'sigma_noise_soma': 25. * pamp,
 'sigma_noise_trunk': 20. * pamp,
 't_AMPA_decay_pathX_dist': 2. * msecond,
 't_AMPA_decay_pathY_prox': 2. * msecond,
 't_NMDA_decay_pathX_dist': 60. * msecond,
 't_NMDA_decay_pathY_prox': 60. * msecond,
 'tau_noise_dist': 1. * msecond,
 'tau_noise_prox': 1. * msecond,
 'tau_noise_soma': 1. * msecond,
 'tau_noise_trunk': 1. * msecond,
 'w_AMPA_pathX_dist': 1.0,
 'w_AMPA_pathY_prox': 1.0,
 'w_NMDA_pathX_dist': 1.0,
 'w_NMDA_pathY_prox': 1.0}


EVENTS
------
['spike_Na_trunk', 'spike_Na_prox', 'spike_Na_dist']


EVENT CONDITIONS
----------------
{'spike_Na_dist': 'V_dist >= Vth_Na_dist and t_in_timesteps >= spiketime_Na_dist +␣
↪refractory_Na_dist * gate_Na_dist',
 'spike_Na_prox': 'V_prox >= Vth_Na_prox and t_in_timesteps >= spiketime_Na_prox +␣
↪refractory_Na_prox * gate_Na_prox',
 'spike_Na_trunk': 'V_trunk >= Vth_Na_trunk and t_in_timesteps >= spiketime_Na_trunk +␣
↪refractory_Na_trunk * '
                   'gate_Na_trunk'}
```

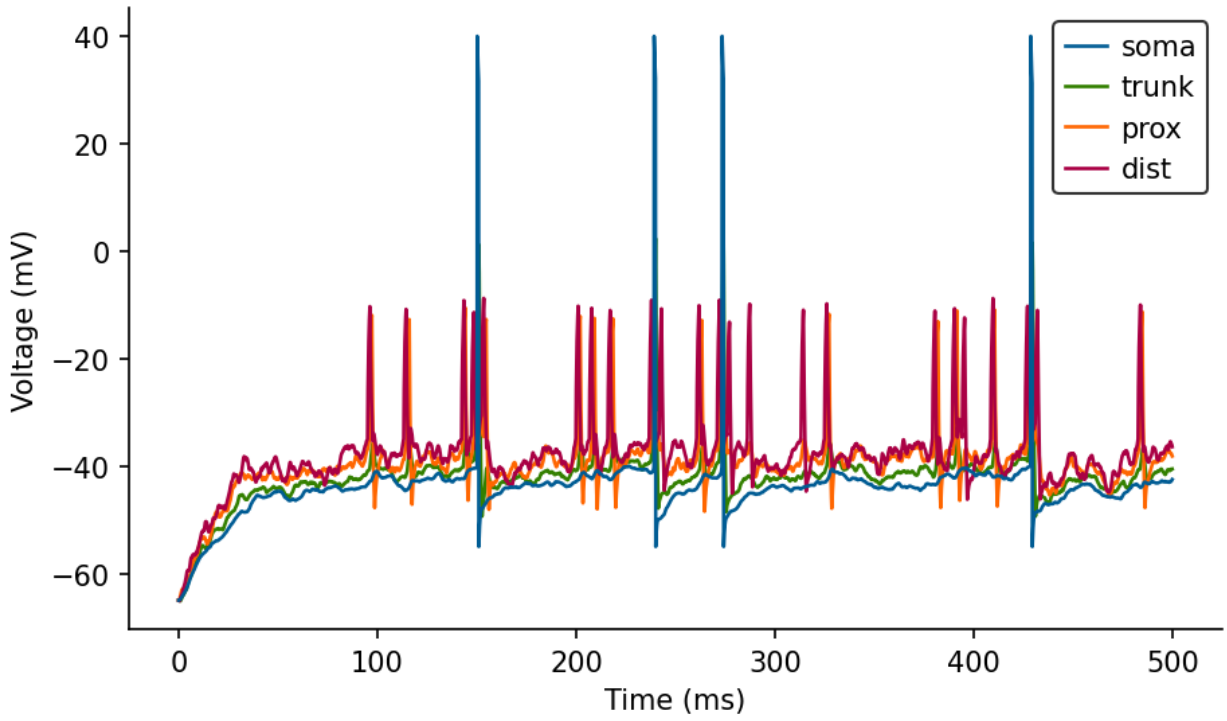
```
# run simulation
net = b.Network(neuron, ap_reset, M)
net.run(500*ms)
```

```
# @title Plot voltages
fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(M.t/ms, M.V_soma[0]/mV, label='soma', zorder=3)
ax.plot(M.t/ms, M.V_trunk[0]/mV, label='trunk')
ax.plot(M.t/ms, M.V_prox[0]/mV, label='prox')
ax.plot(M.t/ms, M.V_dist[0]/mV, label='dist')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('Time (ms)')
ax.set_ylabel('Voltage (mV)')
ax.legend(loc='best');
```



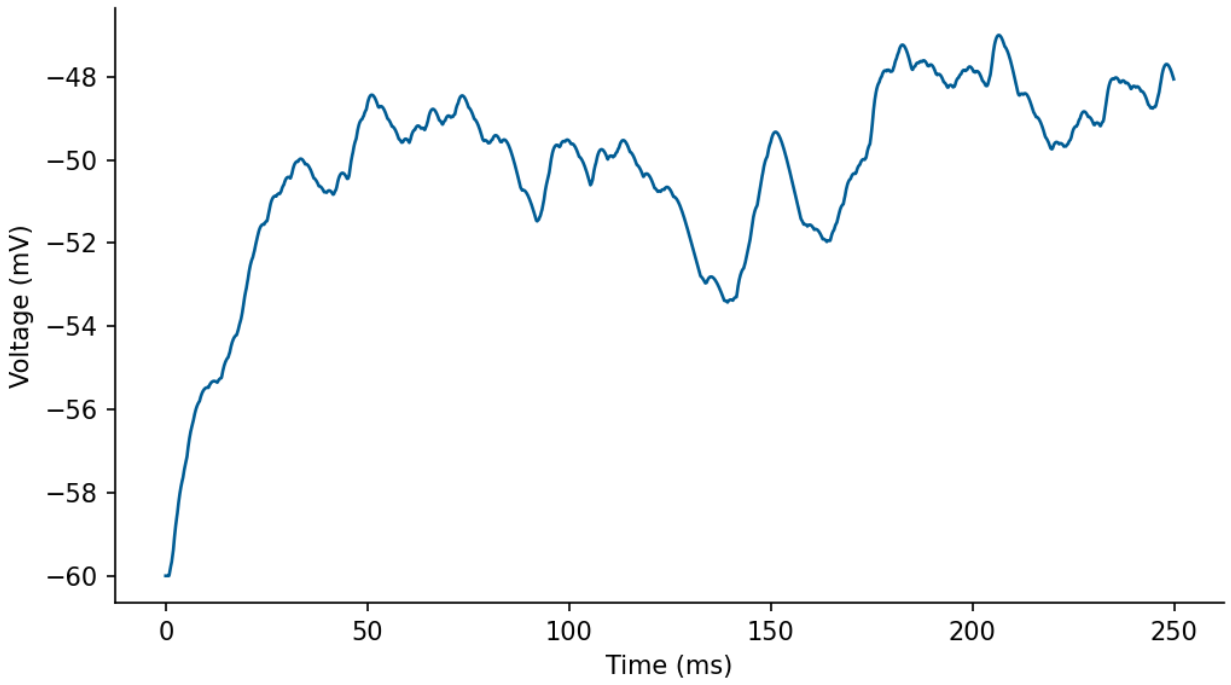
3.8 Point neurons

```
from dendrify import PointNeuronModel

b.start_scope()

# create a point-neuron model and add some Poisson input
point_model = PointNeuronModel(model='leakyIF', v_rest=-60*mV,
                               cm_abs=200*pF, gl_abs=10*nS)
point_model.synapse('AMPA', tag='x', g=1*nS, t_decay=2*ms)
point_neuron = point_model.make_neurongroup(1, method='euler')  # no spiking

Input = b.PoissonGroup(10, rates=100*Hz)
S = b.Synapses(Input, point_neuron, on_pre='s_AMPA_x += 1')
S.connect(p=1)

# monitor
M2 = b.StateMonitor(point_neuron, 'V', record=True)

# simulation
b.run(250*ms)
```

(continues on next page)

(continued from previous page)

```
# plot
fig, ax = plt.subplots(1, 1, figsize=[7, 4])
ax.plot(M2.t/ms, M2.V[0]/mV)
ax.set_xlabel('Time (ms)')
ax.set_ylabel('Voltage (mV)')
fig.tight_layout();
```



CHAPTER

FOUR

COMPARTMENTAL MODELS

4.1 Understanding dSpikes

Dendrify introduces a new event-driven mechanism for modeling dendritic spiking, which is significantly simpler and more efficient than the traditional Hodgkin-Huxley formalism. This mechanism has three distinct phases.

INACTIVE PHASE: When the dendritic voltage is subthreshold OR the simulation step is within the refractory period. dSpikes cannot be generated during this phase.

RISE PHASE: When the dendritic voltage crosses the dSpike threshold AND the refractory period has elapsed. This triggers the instant activation of a positive current that is deactivated after a specified amount of time (`duration_rise`). Also a new refractory period begins.

FALL PHASE: This phase starts automatically with a delay (`offset_fall`) after the dSpike threshold is crossed. A negative current is activated instantly and then is deactivated after a specified amount of time (`duration_fall`).

In this example we show:

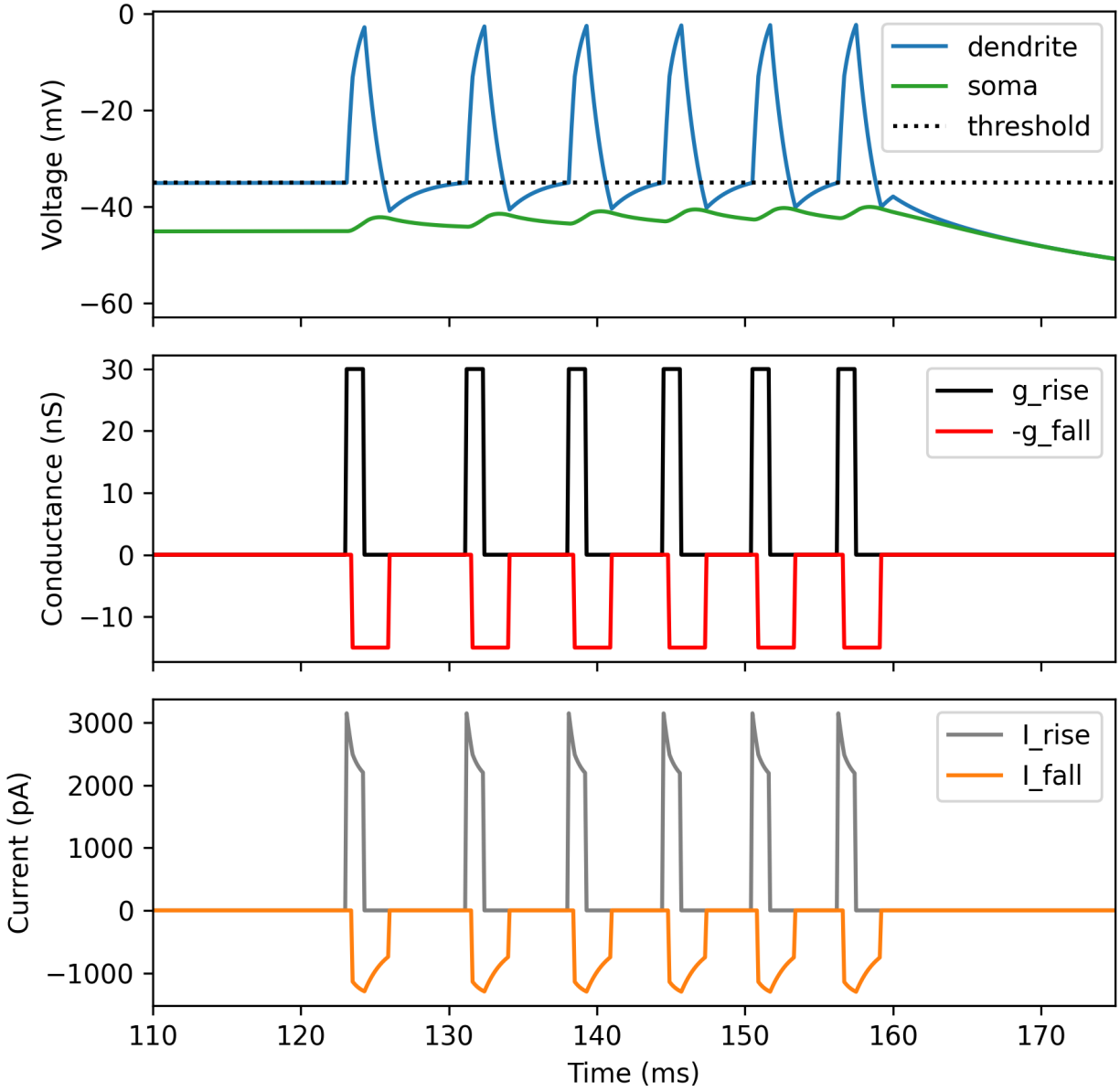
- How dendritic spiking is implemented in Dendrify.

```
import brian2 as b
from brian2.units import ms, mV, nS, pA, pF

from dendrify import Dendrite, NeuronModel, Soma

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create neuron model
soma = Soma('soma', cm_abs=200*pF, gl_abs=10*nS)
dend = Dendrite('dend', cm_abs=50*pF, gl_abs=2.5*nS)
dend.dspikes('Na', g_rise=30*nS, g_fall=15*nS)

model = NeuronModel([(soma, dend, 15*nS)], v_rest=-60*mV)
model.config_dspikes('Na', threshold=-35*mV,
                     duration_rise=1.2*ms, duration_fall=2.4*ms,
                     offset_fall=0.5*ms, refractory=5*ms,
                     reversal_rise='E_Na', reversal_fall='E_K')

# Create neuron group
neuron = model.make_neurongroup(1, method='euler')

# Record variables of interest
vars = ['V_soma', 'V_dend', 'g_rise_Na_dend', 'g_fall_Na_dend',
```

(continues on next page)

(continued from previous page)

```
        'I_rise_Na_dend', 'I_fall_Na_dend']
M = b.StateMonitor(neuron, vars, record=True)

# Run simulation
b.run(10*ms)
neuron.I_ext_dend = 213*pA
b.run(150*ms)
neuron.I_ext_dend = 0*pA
b.run(80*ms)

# Visualize results
time = M.t/ms
v1 = M.V_soma[0]/mV
v2 = M.V_dend[0]/mV

fig, axes = b.subplots(3, 1, figsize=(6, 6), sharex=True)
ax0, ax1, ax2 = axes

ax0.plot(time, v2, label='dendrite')
ax0.plot(time, v1, label='soma', c='C2')
ax0.axhline(-35, ls=':', c='black', label='threshold')
ax0.set_ylabel('Voltage (mV)')
ax0.set_xlim(110, 175)

ax1.plot(time, M.g_rise_Na_dend[0]/nS, label='g_rise', c='black')
ax1.plot(time, -M.g_fall_Na_dend[0]/nS, label='-g_fall', c='red')
ax1.set_ylabel('Conductance (nS)')

ax2.plot(time, M.I_rise_Na_dend[0]/pA, label='I_rise', c='gray')
ax2.plot(time, M.I_fall_Na_dend[0]/pA, label='I_fall', c='C1')
ax2.set_ylabel('Current (pA)')
ax2.set_xlabel('Time (ms)')

for ax in axes:
    ax.legend()
fig.tight_layout()
b.show()
```



4.2 Back-propagating dSpikes

An important property of biological neurons is that action potentials (APs) initiated in the axon can invade the soma and nearby dendrites and propagate backwards toward the dendritic tips. The transmission efficacy of these back-propagating action potentials (bAPs) relies on the dendritic morphology and the presence of dendritic voltage-gated ion channels.

In Dendrify, to achieve this behavior one needs to first recreate a more realistic somatic AP shape by using the `second_reset` and `spike_width` arguments in `make_neurongroup`. In this way, the somatic voltage can be first reset to a more positive value and then below threshold. This allows the passive depolarization of proximal dendrites in responses to somatic APs. If dendrites are also equipped with active ionic mechanisms, this depolarization can trigger the spontaneous generation of dendritic bAPs.

In this example we show:

- How to implement back-propagating dSpikes in Dendrify.
- How to achieve a more realistic somatic AP shape in I&F models, that is essential for the generation of bAPs.

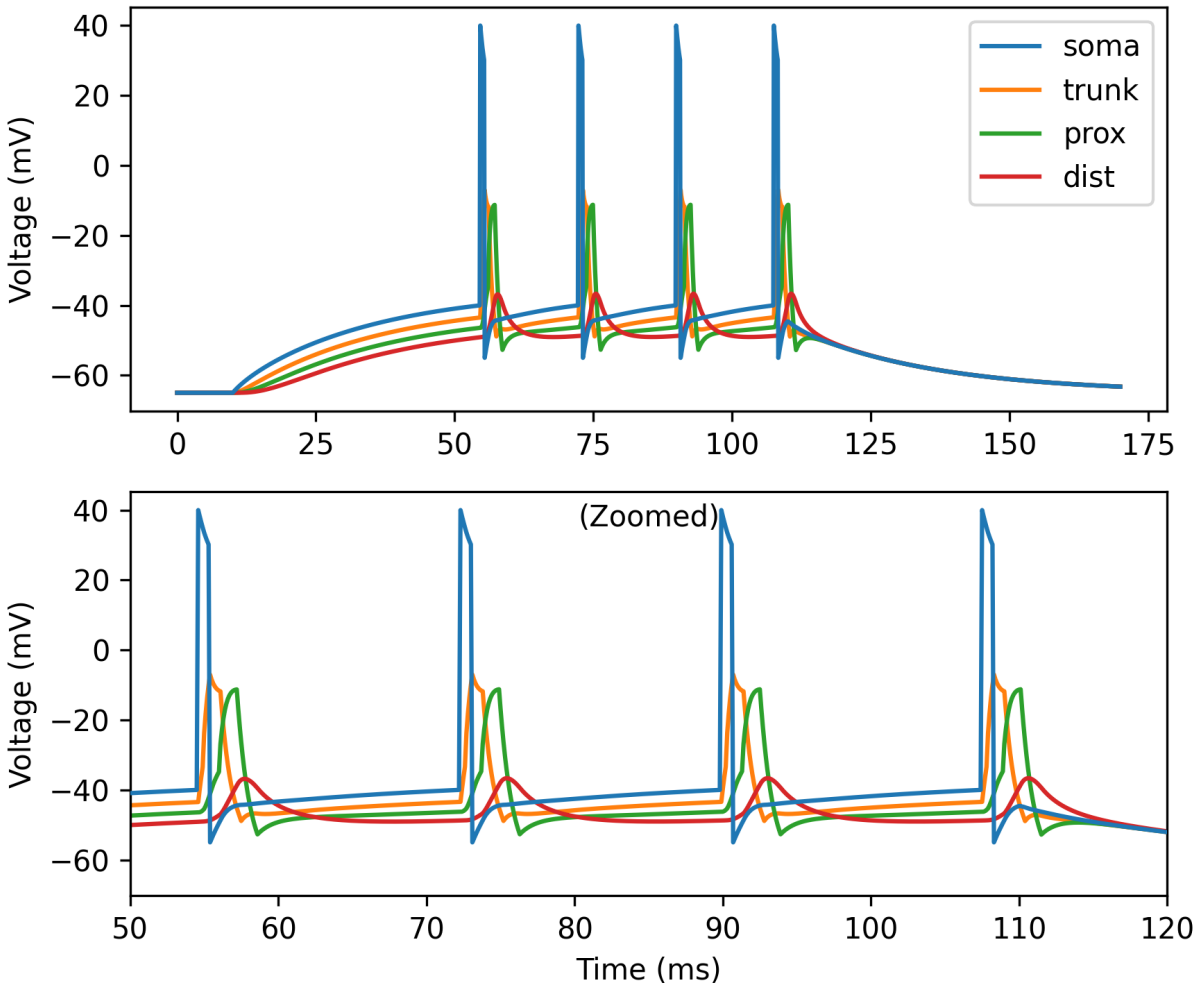
Important: Notice that when a `second_reset` is used, the `make_neurongroup` method returns an additional object which is Brian's Synapses. If your simulation code uses Brian's Networks feature, this additional object should be added to the network as well (also shown in the example below).

```
import brian2 as b
from brian2.units import cm, ms, mV, nS, ohm, pA, uF, um, uS

from dendrify import Dendrite, NeuronModel, Soma

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create neuron model
soma = Soma('soma', model='leakyIF', length=25*um, diameter=25*um)
trunk = Dendrite('trunk', length=100*um, diameter=2.5*um)
prox = Dendrite('prox', length=100*um, diameter=1*um)
dist = Dendrite('dist', length=100*um, diameter=0.5*um)

trunk.dspikes('Na', g_rise=22*nS, g_fall=14*nS)
prox.dspikes('Na', g_rise=9*nS, g_fall=5.7*nS)
dist.dspikes('Na', g_rise=3.7*nS, g_fall=2.4*nS)

con = [(soma, trunk, 15*nS), (trunk, prox, 6*nS), (prox, dist, 2*nS)]
model = NeuronModel(con, cm=1*uF/(cm**2), gl=40*uS/(cm**2),
                    v_rest=-65*mV, r_axial=150*ohm*cm,
                    scale_factor=2.8, spine_factor=1.5)
model.config_dspikes('Na', threshold=-35*mV,
                     duration_rise=1.2*ms, duration_fall=2.4*ms,
                     offset_fall=0.2*ms, refractory=5*ms,
                     reversal_rise='E_Na', reversal_fall='E_K')

# Make a new neurongroup
neuron, ap_reset = model.make_neurongroup(1, method='euler',
                                          threshold='V_soma > -40*mV',
                                          reset='V_soma = 40*mV',
                                          second_reset='V_soma=-55*mV',
                                          spike_width=0.8*ms,
                                          refractory=4*ms)

# Record voltages
vars = ['V_soma', 'V_trunk', 'V_prox', 'V_dist']
M = b.StateMonitor(neuron, vars, record=True)

# Run simulation
net = b.Network(neuron, ap_reset, M)
net.run(10*ms)
neuron.I_ext_soma = 150*pA
net.run(100*ms)
```

(continues on next page)

(continued from previous page)

```
neuron.I_ext_soma = 0*pA
net.run(60*ms)

# Visualize results
fig, axes = b.subplots(2, 1, figsize=(6, 5))
ax0, ax1 = axes

ax0.plot(M.t/ms, M.V_soma[0]/mV, label='soma', zorder=3)
ax0.plot(M.t/ms, M.V_trunk[0]/mV, label='trunk')
ax0.plot(M.t/ms, M.V_prox[0]/mV, label='prox')
ax0.plot(M.t/ms, M.V_dist[0]/mV, label='dist')
ax0.set_ylabel('Voltage (mV)')
ax0.legend()

ax1.plot(M.t/ms, M.V_soma[0]/mV, zorder=3)
ax1.plot(M.t/ms, M.V_trunk[0]/mV)
ax1.plot(M.t/ms, M.V_prox[0]/mV)
ax1.plot(M.t/ms, M.V_dist[0]/mV)
ax1.set_title('(Zoomed)', y=1, pad=-12, fontsize=10)
ax1.set_xlabel('Time (ms)')
ax1.set_ylabel('Voltage (mV)')
ax1.set_xlim(50, 120)

fig.tight_layout()
b.show()
```



4.3 Active vs passive dendrites

In pyramidal neurons, distal synapses have often a minute effect on the somatic membrane potential due to strong dendritic attenuation. However, the activation of dendritic spikes can amplify synaptic inputs that are temporally correlated, increasing the probability of somatic AP generation.

In this example we show:

- How to create a compartmental model with passive or active dendrites.
- How dendritic spiking may affect somatic AP generation.

```
import brian2 as b
from brian2.units import Hz, ms, mV, nS, pF

from dendrify import Dendrite, NeuronModel, Soma

b.prefs.codegen.target = 'numpy'  # faster for simple simulations
```

(continues on next page)

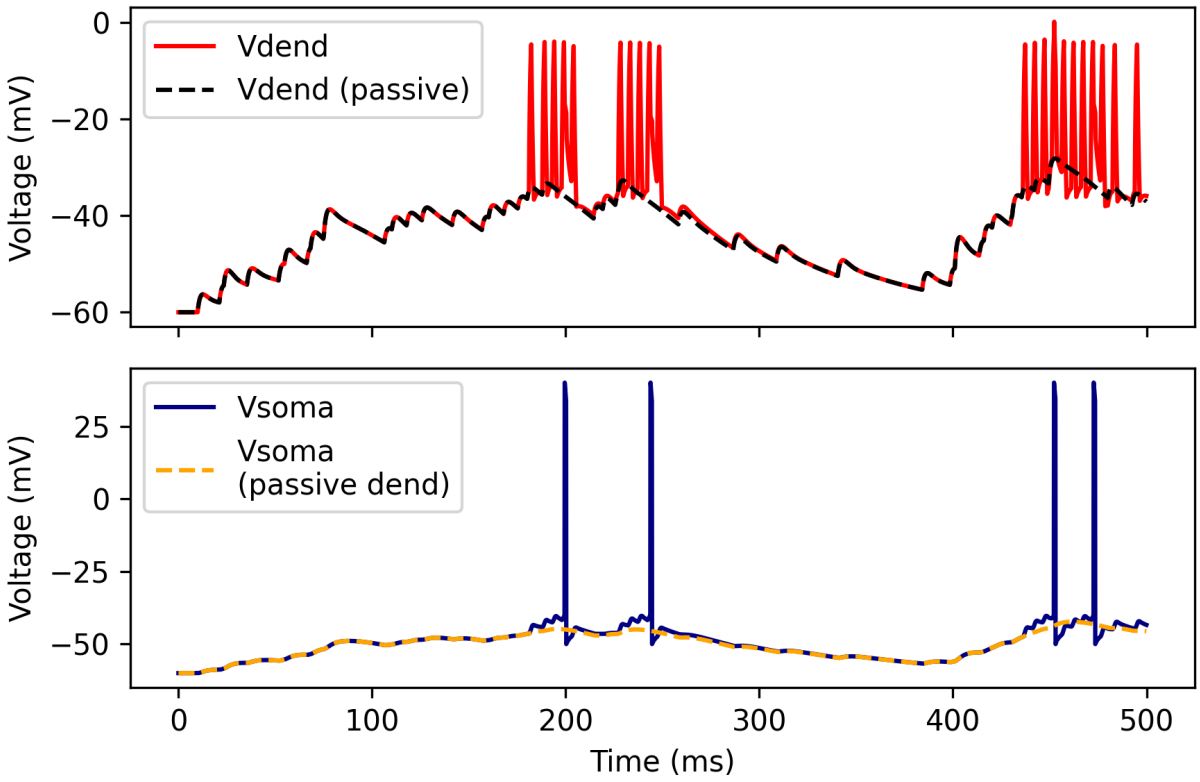
(continued from previous page)

```
# Create neuron model with passive dendrites
soma = Soma('soma', cm_abs=200*pF, gl_abs=10*nS)
dend = Dendrite('dend', cm_abs=50*pF, gl_abs=2.5*nS)
dend.synapse('AMPA', tag='x', g=3*nS,  t_decay=2*ms)
dend.synapse('NMDA', tag='x', g=3*nS,  t_decay=60*ms)
model_passive = NeuronModel([(soma, dend, 15*nS)], v_rest=-60*mV)

# Add dendritic spikes and create a neuron model with active dendrites
dend.dspikes('Na', g_rise=30*nS, g_fall=14*nS)
model_active = NeuronModel([(soma, dend, 15*nS)], v_rest=-60*mV)
model_active.config_dspikes(
    'Na', threshold=-35*mV,
    duration_rise=1.2*ms, duration_fall=2.4*ms,
    offset_fall=0.2*ms, refractory=5*ms,
    reversal_rise='E_Na', reversal_fall='E_K')

# Create a neuron group with passive dendrites
neuron_passive, reset_p = model_passive.make_neurongroup(
    1, method='euler',
    threshold='V_soma > -40*mV',
    reset='V_soma = 40*mV',
    second_reset='V_soma=-50*mV',
    spike_width=0.8*ms,
    refractory=4*ms)

# Create a neuron group with active dendrites
neuron_active, reset_a = model_active.make_neurongroup(
    1, method='euler',
    threshold='V_soma > -40*mV',
    reset='V_soma = 40*mV',
    second_reset='V_soma=-50*mV',
    spike_width=0.8*ms,
    refractory=4*ms)

# Create random Poisson input
Input_p = b.PoissonGroup(5, rates=20*Hz)
Input_a = b.PoissonGroup(5, rates=20*Hz)

# Create synapses
S_p = b.Synapses(Input_p, neuron_passive,
                 on_pre='s_AMPA_x_dend += 1; s_NMDA_x_dend += 1')
S_p.connect(p=1)

S_a = b.Synapses(Input_a, neuron_active,
                 on_pre='s_AMPA_x_dend += 1; s_NMDA_x_dend += 1')
S_a.connect(p=1)

# Record voltages
vars = ['V_soma', 'V_dend']
M_p = b.StateMonitor(neuron_passive, vars, record=True)
M_a = b.StateMonitor(neuron_active, vars, record=True)
```

(continues on next page)

(continued from previous page)

```
# Run simulation
b.seed(123)  # for reproducibility
net_passive = b.Network(neuron_passive, reset_p, Input_p, S_p, M_p)
net_passive.run(500*ms)
b.start_scope()  # clear previous simulation
b.seed(123)  # for reproducibility
net_active = b.Network(neuron_active, reset_a, Input_a, S_a, M_a)
net_active.run(500*ms)

# Visualize results
time_p = M_p.t/ms
vs_p = M_p.V_soma[0]/mV
vd_p = M_p.V_dend[0]/mV
time_a = M_a.t/ms
vs_a = M_a.V_soma[0]/mV
vd_a = M_a.V_dend[0]/mV

fig, axes = b.subplots(2, 1, figsize=(6, 4), sharex=True)
ax0, ax1 = axes
ax0.plot(time_a, vd_a, label='Vdend', c='red')
ax0.plot(time_p, vd_p, '--', label='Vdend (passive)', c='black')
ax0.set_ylabel('Voltage (mV)')
ax0.legend(loc=2)

ax1.plot(time_a, vs_a, label='Vsoma', c='navy')
ax1.plot(time_p, vs_p, '--', label='Vsoma\n(passive dend)', c='orange')
ax1.set_xlabel('Time (ms)')
ax1.set_ylabel('Voltage (mV)')
ax1.legend(loc=2)

fig.tight_layout()
b.show()
```



4.4 Networks of compartmental neurons

In this example we show:

- How to create a recurrent network of compartmental neurons.
- How active dendrites can alter network responses.

```
import brian2 as b
from brian2.units import Hz, ms, mV, nS, pF

from dendrify import Dendrite, NeuronModel, Soma

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create neuron model with passive dendrites
soma = Soma('soma', cm_abs=200*pF, gl_abs=10*nS)
dend = Dendrite('dend', cm_abs=50*pF, gl_abs=2.5*nS)
dend.synapse('AMPA', tag='external', g=1*nS,  t_decay=5*ms)
dend.synapse('AMPA', tag='recurrent', g=1*nS,  t_decay=5*ms)
model_passive = NeuronModel([(soma, dend, 15*nS)], v_rest=-60*mV)

# Add dendritic spikes and create a neuron model with active dendrites
dend.dspikes('Na', g_rise=30*nS, g_fall=14*nS)
model_active = NeuronModel([(soma, dend, 15*nS)], v_rest=-60*mV)
```

(continues on next page)

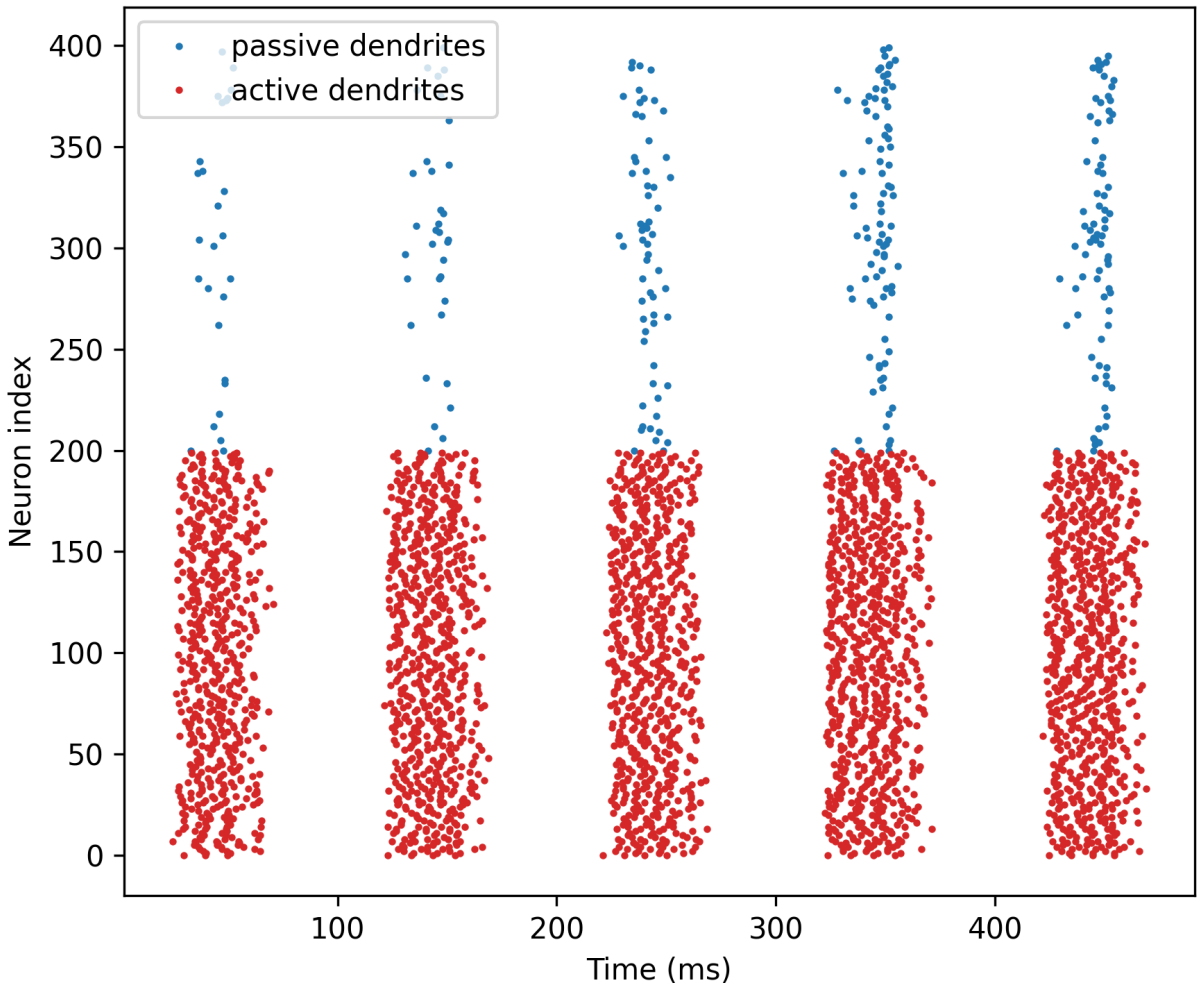
(continued from previous page)

```
model_active.config_dspikes(
    'Na', threshold=-35*mV,
    duration_rise=1.2*ms, duration_fall=2.4*ms,
    offset_fall=0.2*ms, refractory=5*ms,
    reversal_rise='E_Na', reversal_fall='E_K')

# Create a neuron group with passive dendrites
neuron_passive, reset_p = model_passive.make_neurongroup(
    200, method='euler',
    threshold='V_soma > -40*mV',
    reset='V_soma = 40*mV',
    second_reset='V_soma=-50*mV',
    spike_width=0.8*ms,
    refractory=4*ms)

# Create a neuron group with active dendrites
neuron_active, reset_a = model_active.make_neurongroup(
    200, method='euler',
    threshold='V_soma > -40*mV',
    reset='V_soma = 40*mV',
    second_reset='V_soma=-50*mV',
    spike_width=0.8*ms,
    refractory=4*ms)

# Create random Poisson input
# Protocol: five 50 ms blocks of 50 Hz stimulation, followed by 50 ms of silence
stimulus = b.TimedArray(b.tile([50., 0.], 5)*Hz, dt=50.*ms)
Input_a = b.PoissonGroup(200, rates='stimulus(t)')
Input_p = b.PoissonGroup(200, rates='stimulus(t)')

# Create synapses for external input
S_input_passive = b.Synapses(
    Input_p, neuron_passive,
    on_pre='s_AMPA_external_dend += 1')
S_input_passive.connect(p=0.2)

S_input_active = b.Synapses(
    Input_a, neuron_active,
    on_pre='s_AMPA_external_dend += 1')
S_input_active.connect(p=0.2)

# Create recurrent synapses
S_recurrent_passive = b.Synapses(
    neuron_passive, neuron_passive,
    on_pre='s_AMPA_recurrent_dend += 1')
S_recurrent_passive.connect(p=0.1)

S_recurrent_active = b.Synapses(
    neuron_active, neuron_active,
    on_pre='s_AMPA_recurrent_dend += 1')
S_recurrent_active.connect(p=0.1)
```

(continues on next page)

(continued from previous page)

```
# Record spikes
spikes_passive = b.SpikeMonitor(neuron_passive)
spikes_active = b.SpikeMonitor(neuron_active)

# Run simulation
b.seed(123)  # for reproducibility
net_passive = b.Network(neuron_passive, reset_p, Input_p,
                        S_input_passive, S_recurrent_passive,
                        spikes_passive)
net_passive.run(550*ms)
b.start_scope()  # clear previous simulation
b.seed(123)  # for reproducibility
net_active = b.Network(neuron_active, reset_a, Input_a,
                       S_input_active, S_recurrent_active,
                       spikes_active)
net_active.run(550*ms)

# Visualize results
b.figure(figsize=[6, 5])
b.plot(spikes_passive.t/ms, spikes_passive.i + 200,
       '.', ms=3, label='passive dendrites')
b.plot(spikes_active.t/ms, spikes_active.i,
       '.', ms=3, c='C3', label='active dendrites')
b.xlabel('Time (ms)')
b.ylabel('Neuron index')
b.legend()
b.tight_layout()
b.show()
```


passive dendrites
active dendrites
Neuron index
Time (ms)
0
50
100
150
200
250
300
350
400
100
200
300
400

CHAPTER

FIVE

POINT-NEURON MODELS

5.1 AdEx neuron

The Dendrify implementation of the Adaptive exponential integrate-and-fire model (adapted from Brian's examples).

Resources:

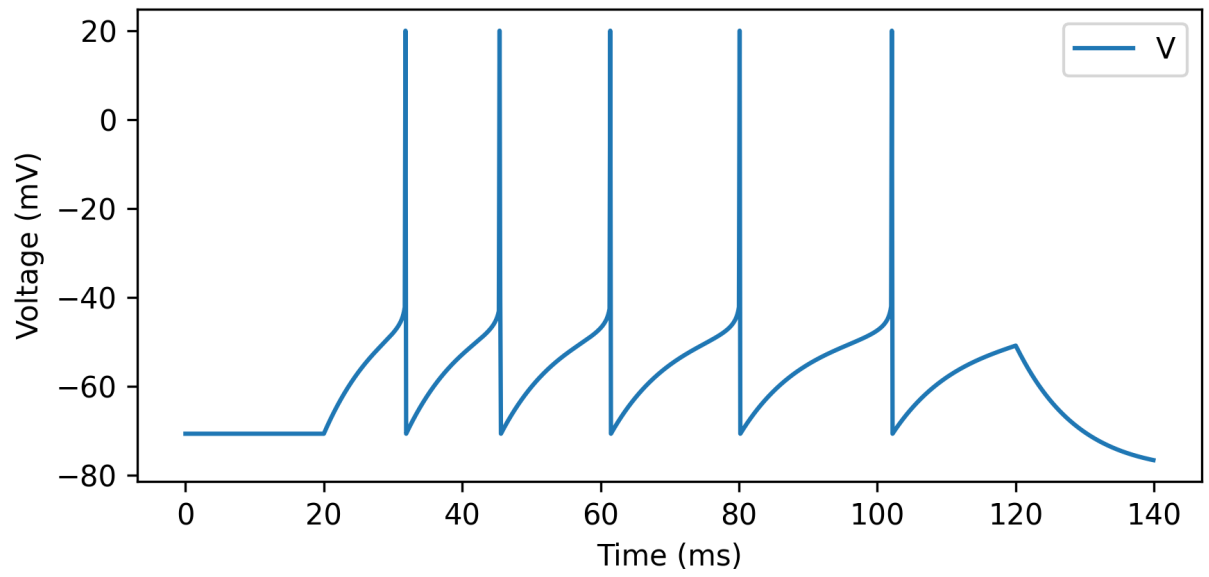
- http://www.scholarpedia.org/article/Adaptive_exponential_integrate-and-fire_model
- https://pubmed.ncbi.nlm.nih.gov/16014787/

```
import brian2 as b
from brian2.units import ms, mV, nA, nS, pF

from dendrify import PointNeuronModel

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create neuron model
model = PointNeuronModel(model='adex',
                         cm_abs=281*pF,
                         gl_abs=30*nS,
                         v_rest=-70.6*mV)

# Include adex parameters
model.add_params({'Vth': -50.4*mV,
                  'DeltaT': 2*mV,
                  'tauw': 144*ms,
                  'a': 4*nS,
                  'b': 0.0805*nA,
                  'Vr': -70.6*mV})

# Create a NeuronGroup
neuron = model.make_neurongroup(N=1, threshold='V>Vth+5*DeltaT',
                                reset='V=Vr; w+=b',
                                method='euler')

# Record voltages and spike times
trace = b.StateMonitor(neuron, 'V', record=True)
spikes = b.SpikeMonitor(neuron)

# Run simulation
```

(continues on next page)

(continued from previous page)

```
b.run(20 * ms)
neuron.I_ext = 1*nA
b.run(100 * ms)
neuron.I_ext = 0*nA
b.run(20 * ms)

# Trick to draw nicer spikes in I&F models
vm = trace[0].V[:]
for t in spikes.t:
    i = int(t / b.defaultclock.dt)
    vm[i] = 20*mV

# Plot results
b.figure(figsize=[6, 3])
b.plot(trace.t / ms, vm / mV, label='V')
b.xlabel('Time (ms)')
b.ylabel('Voltage (mV)')
b.legend()
b.tight_layout()
b.show()
```



5.2 AdEx neuron + noise

The Dendrify implementation of the Adaptive exponential integrate-and-fire model (adapted from Brian's examples).

In this example, we also explore:

- How to add gaussian noise.
- How to create NeuronGroups with different properties using a single PointNeuronModel.

Resources:

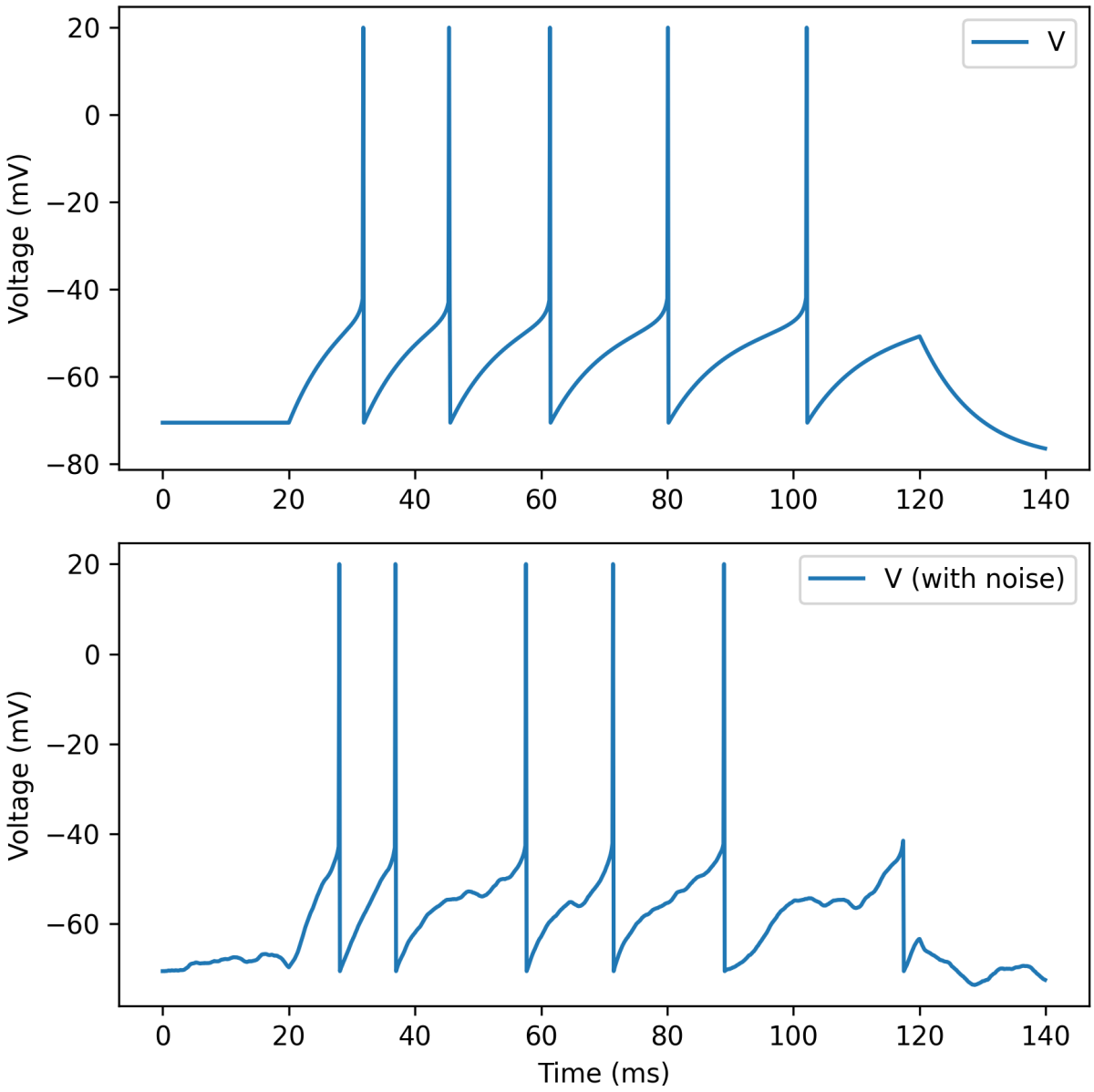
- http://www.scholarpedia.org/article/Adaptive_exponential_integrate-and-fire_model
- https://pubmed.ncbi.nlm.nih.gov/16014787/

```
import brian2 as b
from brian2.units import ms, mV, nA, nS, pA, pF

from dendrify import PointNeuronModel

b.prefs.codegen.target = 'numpy'  # faster for simple simulations
b.seed(1234)  # for reproducibility

# Create neuron model
model = PointNeuronModel(model='adex',
                         cm_abs=281*pF,
                         gl_abs=30*nS,
                         v_rest=-70.6*mV)

model.add_params({'Vth': -50.4*mV,
                  'DeltaT': 2*mV,
                  'tauw': 144*ms,
                  'a': 4*nS,
                  'b': 0.0805*nA,
                  'Vr': -70.6*mV})

# Create a NeuronGroup
neuron = model.make_neurongroup(N=1, threshold='V>Vth+5*DeltaT',
                                reset='V=Vr; w+=b',
                                method='euler')

# Update model with noise and create a new NeuronGroup
model.noise(mean=50*pA, sigma=300*pA, tau=2*ms)
noisy_neuron = model.make_neurongroup(N=1, threshold='V>Vth+5*DeltaT',
                                      reset='V=Vr; w+=b',
                                      method='euler')

# Record voltages and spike times
trace = b.StateMonitor(neuron, 'V', record=True)
spikes = b.SpikeMonitor(neuron)
noisy_trace = b.StateMonitor(noisy_neuron, 'V', record=True)
noisy_spikes = b.SpikeMonitor(noisy_neuron)

# Run simulation
b.run(20 * ms)
neuron.I_ext = 1*nA
noisy_neuron.I_ext = 1*nA
b.run(100 * ms)
neuron.I_ext = 0*nA
noisy_neuron.I_ext = 0*nA
b.run(20 * ms)

# Trick to draw nicer spikes in I&F models
vm = trace[0].V[:]
noisy_vm = noisy_trace[0].V[:]
```

(continues on next page)

(continued from previous page)

```
for t1, t2 in zip(spikes.t, noisy_spikes.t):
    i = int(t1 / b.defaultclock.dt)
    j = int(t2 / b.defaultclock.dt)
    vm[i] = 20*mV
    noisy_vm[j] = 20*mV

# Plot results
fig, axes = b.subplots(2, 1, figsize=[6, 6])
ax1, ax2 = axes
ax1.plot(trace.t / ms, vm / mV, label='V')
ax1.set_ylabel('Voltage (mV)')
ax1.legend()

ax2.plot(noisy_trace.t / ms, noisy_vm / mV, label='V (with noise)')
ax2.set_ylabel('Voltage (mV)')
ax2.set_xlabel('Time (ms)')
ax2.legend()
fig.tight_layout()
b.show()
```



5.3 AdEx network + synapses

The Dendrify implementation of the Adaptive exponential integrate-and-fire model (adapted from Brian's examples).

In this example, we also explore:

- How to add random Poisson synaptic input.
- How to create a basic network model.

Resources:

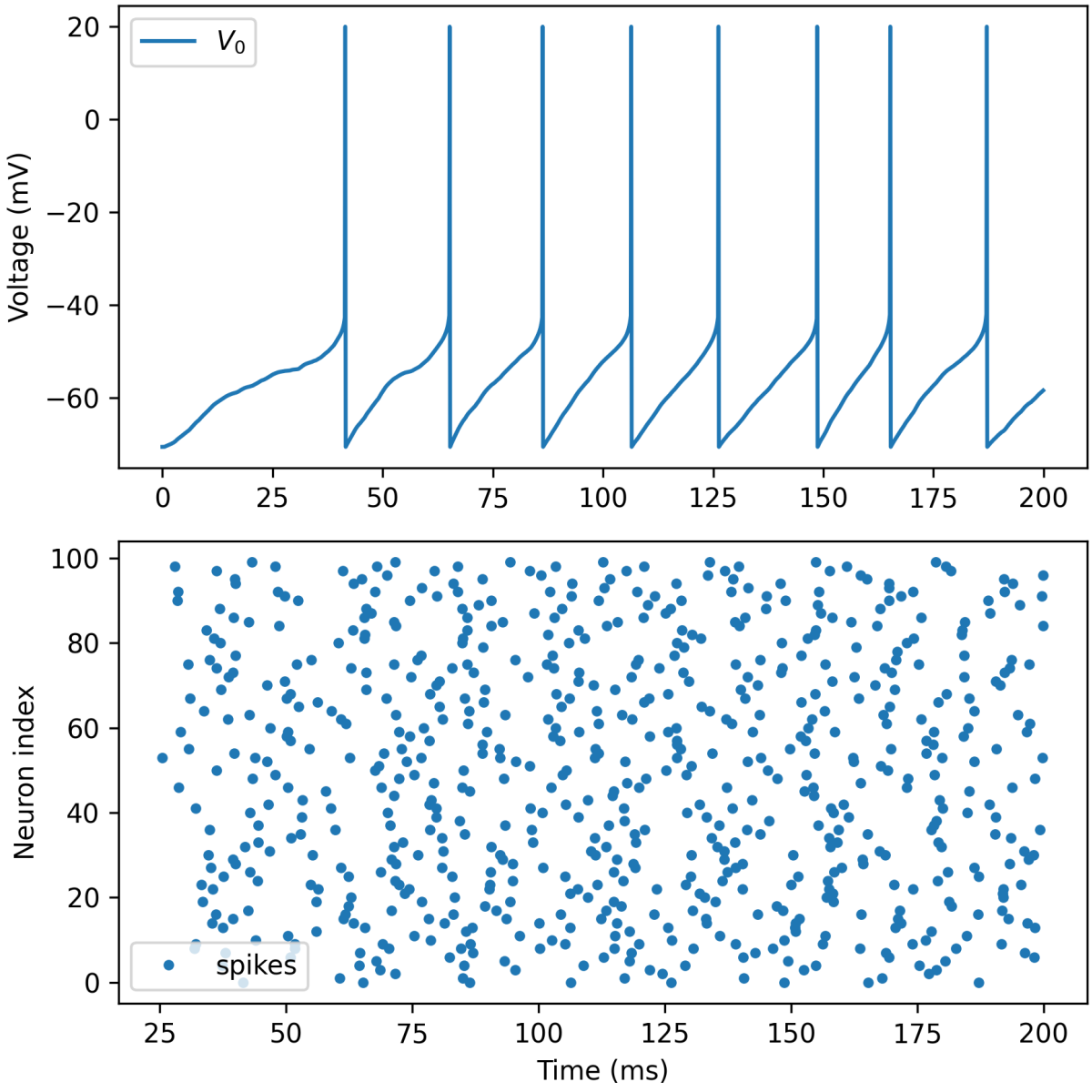
- http://www.scholarpedia.org/article/Adaptive_exponential_integrate-and-fire_model
- https://pubmed.ncbi.nlm.nih.gov/16014787/

```
import brian2 as b
from brian2.units import Hz, ms, mV, nA, nS, pF

from dendrify import PointNeuronModel

b.prefs.codegen.target = 'numpy'  # faster for simple simulations
b.seed(1234)  # for reproducibility

# Create neuron model and add AMPA equations
model = PointNeuronModel(model='adex',
                         cm_abs=281*pF,
                         gl_abs=30*nS,
                         v_rest=-70.6*mV)
model.synapse('AMPA', tag='x', g=0.5*nS, t_decay=2*ms)
model.synapse('NMDA', tag='x', g=0.5*nS, t_decay=50*ms)

# Include adex parameters
model.add_params({'Vth': -50.4*mV,
                  'DeltaT': 2*mV,
                  'tauw': 144*ms,
                  'a': 4*nS,
                  'b': 0.0805*nA,
                  'Vr': -70.6*mV})

# Create a NeuronGroup
neuron = model.make_neurongroup(N=100, threshold='V>Vth+5*DeltaT',
                                reset='V=Vr; w+=b',
                                method='euler')

# Create a Poisson input
Input = b.PoissonGroup(200, rates=100*Hz)

# Randomly connect Poisson input to NeuronGroup
S = b.Synapses(Input, neuron, on_pre='s_AMPA_x += 1; s_NMDA_x += 1')
S.connect(p=0.25)

# Record voltages and spike times
trace = b.StateMonitor(neuron, 'V', record=0)
spikes = b.SpikeMonitor(neuron)

# Run simulation
b.run(200 * ms)

# Trick to draw nicer spikes in I&F models
vm = trace[0].V[:]
for t in spikes.spike_trains()[0]:
    i = int(t / b.defaultclock.dt)
    vm[i] = 20*mV

# Plot results
fig, axes = b.subplots(2, 1, figsize=[6, 6])
ax1, ax2 = axes
ax1.plot(trace.t / ms, vm / mV, label='$V_0$')
```

(continues on next page)

(continued from previous page)

```
ax1.set_ylabel('Voltage (mV)')
ax1.legend()
ax2.plot(spikes.t/ms, spikes.i, '.', label='spikes')
ax2.set_xlabel('Time (ms)')
ax2.set_ylabel('Neuron index')
ax2.legend()
fig.tight_layout()
b.show()
```



5.4 Other adaptive models

Apart from the Adaptive Exponential integrate-and-fire model (AdEx), Dendrify also supports other types of adaptive models, that can be preferred in some cases. In this example, we compare the behavior of two different adaptive models:

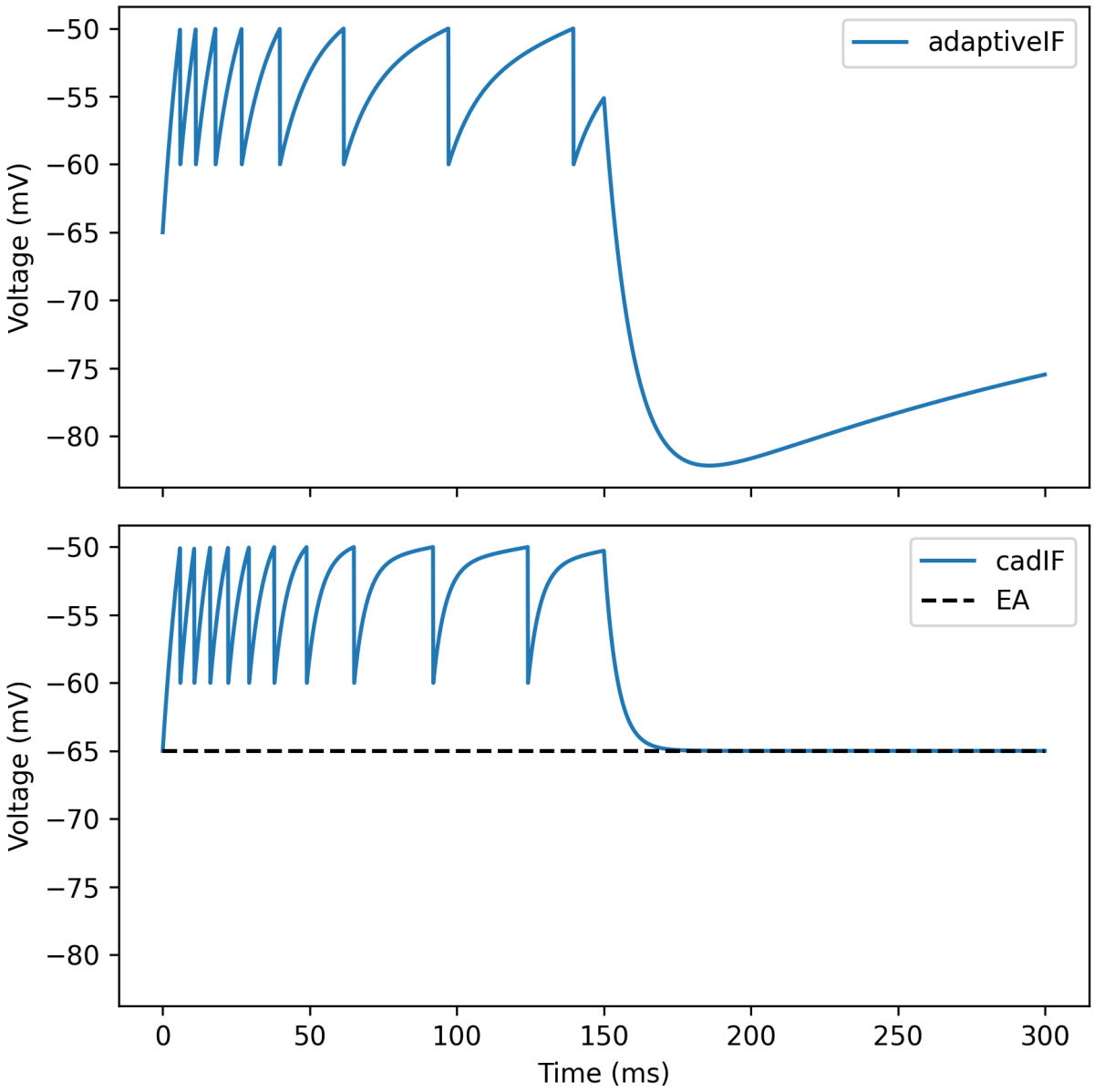
- The Adaptive IF model with current-based adaptation (adaptiveIF), which is simpler and faster to simulate than AdEx.
- The Conductance-based adaptive IF model (cadIF), which provides a solution to a common problem of current-based models, namely the excessive membrane hyperpolarization that can occur after high-frequency firing.

```
import brian2 as b
from brian2.units import Hz, ms, mV, nA, nS, pA, pF

from dendrify import PointNeuronModel

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# A point neuron model with current-based adaptation
adaptiveIF = PointNeuronModel(
    model='adaptiveIF',
    cm_abs=150*pF,
    gl_abs=15*nS,
    v_rest=-65*mV)

adaptiveIF.add_params(
    {'Vth': -50*mV,
     'tauw': 210*ms,
     'a': 0*nS,     # no subthreshold adaptation for simplicity
     'b': 60*pA,
     'Vr': -60*mV})

adaptiveIF_neuron = adaptiveIF.make_neurongroup(
    N=1,
    threshold='V>Vth',
    reset='V=Vr; w+=b',
    method='euler')

# A point neuron model with conductance-based adaptation
cadIF = PointNeuronModel(
    model='cadIF',
    cm_abs=150*pF,
    gl_abs=15*nS,
    v_rest=-65*mV)

cadIF.add_params(
    {'Vth': -50*mV,
     'tauA': 210*ms,
     'gAmax': 0*nS,     # no subthreshold adaptation for simplicity
     'delta_gA': 3*nS,
     'Vr': -60*mV,
     'EA': -65*mV})

cadIF_neuron = cadIF.make_neurongroup(
```

(continues on next page)

(continued from previous page)

```
    N=1,
    threshold='V>Vth',
    reset='V=Vr; gA+=delta_gA',
    method='euler')

# Record voltages
adaptiveIF_trace = b.StateMonitor(adaptiveIF_neuron, ['V'], record=0)
cadIF_trace = b.StateMonitor(cadIF_neuron, ['V'], record=0)

# Run simulation
adaptiveIF_neuron.I_ext = 500*pA
cadIF_neuron.I_ext = 500*pA
b.run(150 * ms)
adaptiveIF_neuron.I_ext = 0*pA
cadIF_neuron.I_ext = 0*pA
b.run(150 * ms)

# Plot results
fig, axes = b.subplots(2, 1, figsize=[6, 6], sharex=True, sharey=True)
ax1, ax2 = axes
ax1.plot(adaptiveIF_trace.t / ms,
         adaptiveIF_trace[0].V / mV,
         label='adaptiveIF')
ax1.set_ylabel('Voltage (mV)')
ax1.legend()

ax2.plot(cadIF_trace.t / ms,
         cadIF_trace[0].V / mV,
         label='cadIF')
ax2.hlines(-65, 0, 300, 'k', '--', label='EA')
ax2.set_xlabel('Time (ms)')
ax2.set_ylabel('Voltage (mV)')
ax2.legend()
fig.tight_layout()
b.show()
```



5.5 LIF network + inhibition

In this example, we present a simple network of generic leaky integrate-and-fire units comprising interconnected excitatory and inhibitory neurons.

In this example, we also explore:

- How to add different types of synaptic equations.
- How to achieve more complex network connectivity.

```
import brian2 as b
from brian2.units import Hz, ms, mV, nS, pF
```

(continues on next page)

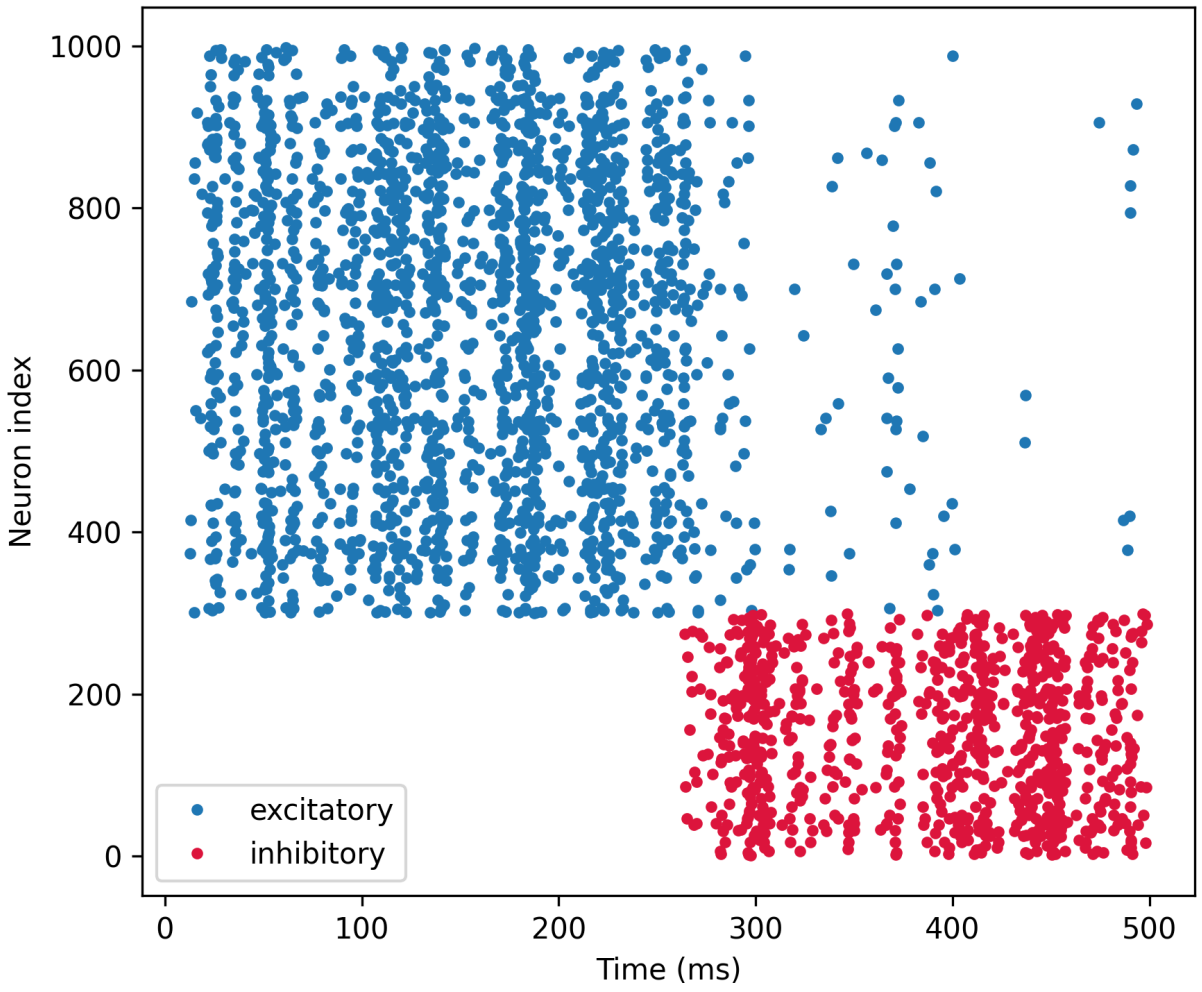
(continued from previous page)

```
from dendrify import PointNeuronModel

b.prefs.codegen.target = 'numpy'  # faster for simple simulations
b.seed(123)  # for reproducibility

N_e = 700
N_i = 300

# Create a neuron model
model = PointNeuronModel(model='leakyIF', cm_abs=281*pF, gl_abs=30*nS,
                         v_rest=-70.6*mV)

# external excitatory input
model.synapse('AMPA', tag='ext', g=2*nS, t_decay=2.5*ms)
model.synapse('GABA', tag='inh', g=2*nS, t_decay=7.5*ms)  # feedback inhibition
model.add_params({'Vth': -40.4*mV, 'Vr': -65.6*mV})

# Create a NeuronGroup
neurons = model.make_neurongroup(N=N_e+N_i, threshold='V>Vth',
                                 reset='V=Vr', method='euler')

# Subpopulation of 300 inhibitory neurons
inhibitory = neurons[:N_i]

# Subpopulation of 700 excitatory neurons
excitatory = neurons[N_i:]

# Create a Poisson input
Input = b.PoissonGroup(200, rates=90*Hz)

# Specify synaptic connections
Syn_ext_a = b.Synapses(Input, excitatory, on_pre='s_AMPA_ext += 1')
Syn_ext_a.connect(p=0.2)

Syn_ext_b = b.Synapses(Input, inhibitory, on_pre='s_AMPA_ext += 1')
Syn_ext_b.connect(p=0)  # initially no connections to inhibitory neurons

Syn_inh = b.Synapses(inhibitory, excitatory, on_pre='s_GABA_inh += 1')
Syn_inh.connect(p=0.15)

# Record voltages and spike times
spikes_e = b.SpikeMonitor(excitatory)
spikes_i = b.SpikeMonitor(inhibitory)

# Run simulation
b.run(250 * ms)
Syn_ext_b.connect(p=0.2)  # add connections to inhibitory neurons
b.run(250 * ms)

# Plot results
b.figure(figsize=[6, 5])
```

(continues on next page)

(continued from previous page)

```
b.plot(spikes_e.t/ms, spikes_e.i+N_i, '.', label='excitatory')
b.plot(spikes_i.t/ms, spikes_i.i, '.', label='inhibitory', c='crimson')
b.xlabel('Time (ms)')
b.ylabel('Neuron index')
b.legend()
b.tight_layout()
b.show()
```



CHAPTER

SIX

SYNAPTIC MODELS

6.1 AMPA synapses

AMPA receptors are the most common type of ionotropic glutamate receptors in the brain. They have fast kinetics and are responsible for excitatory synaptic transmission.

In this example we show:

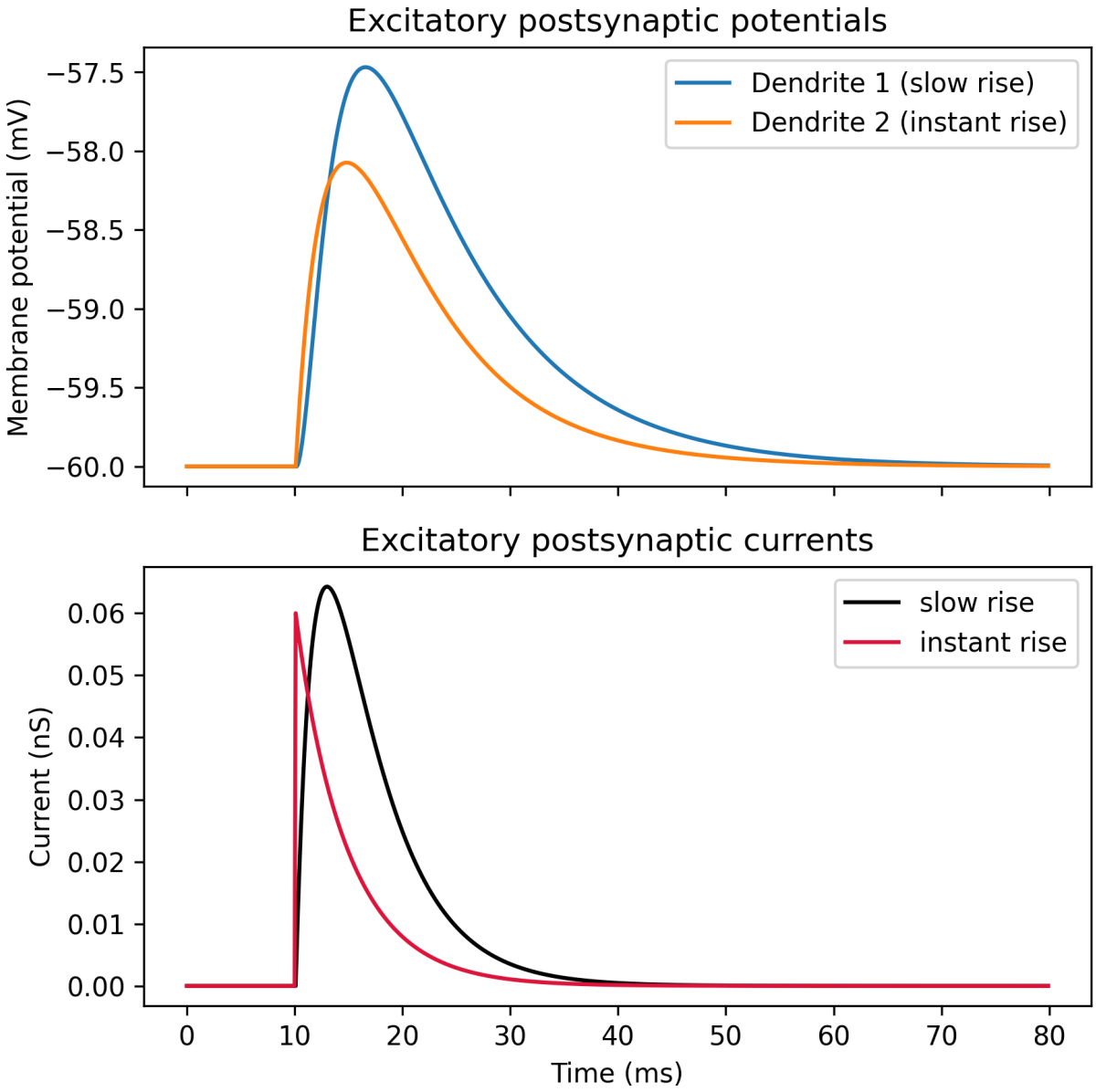
- How to add and activate AMPA synapses at various model compartments.
- How to adjust the synaptic conductance and kinetics of AMPA synapses.

```
import brian2 as b
from brian2.units import ms, mV, nS, pF

from dendrify import Dendrite, NeuronModel, Soma

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create a simple 3-compartment neuron model
soma = Soma('soma', cm_abs=100*pF, gl_abs=10*nS)
dend1 = Dendrite('dend1', cm_abs=50*pF, gl_abs=5*nS)
dend2 = Dendrite('dend2', cm_abs=50*pF, gl_abs=5*nS)

# Add AMPA synapse with rise and decay kinetics to dend1
dend1.synapse('AMPA', tag='slow', g=1*nS, t_rise=2*ms, t_decay=5*ms)

# Add AMPA synapse with instantaneous rise and decay kinetics to dend2
dend2.synapse('AMPA', tag='fast', g=1*nS,  t_decay=5*ms)

# Merge the compartments into a single neuron model
model = NeuronModel([(soma, dend1, 10*nS),
                     (dend1, dend2, 10*nS)],
                    v_rest=-60*mV)

# Create 2 neurons (no somatic spiking for simplicity)
neurons = model.make_neurongroup(2, method='euler')

# Each neuron receives a spike at t=10ms at different dendrites
input_spikes = b.SpikeGeneratorGroup(2, [0, 1], [10, 10]*ms)
slow_ampa = b.Synapses(input_spikes, neurons, on_pre='s_AMPA_slow_dend1 += 1')
slow_ampa.connect(i=0, j=0)
```

(continues on next page)

(continued from previous page)

```
fast_ampa = b.Synapses(input_spikes, neurons, on_pre='s_AMPA_fast_dend2 += 1')
fast_ampa.connect(i=1, j=1)

# Create monitors
variables = ['V_dend1', 'V_dend2', 'I_AMPA_slow_dend1', 'I_AMPA_fast_dend2']
M = b.StateMonitor(neurons, variables, record=True)

# Run the simulation
b.run(80*ms)

# Visualize results
time = M.t/ms
vd1 = M.V_dend1[0]/mV
vd2 = M.V_dend2[1]/mV
ampa1 = M.I_AMPA_slow_dend1[0]/nS
ampa2 = M.I_AMPA_fast_dend2[1]/nS

fig, axes = b.subplots(2, 1, sharex=True, figsize=(6, 6))
ax0, ax1 = axes

ax0.set_title('Excitatory postsynaptic potentials')
ax0.plot(time, vd1, label='Dendrite 1 (slow rise)')
ax0.plot(time, vd2, label='Dendrite 2 (instant rise)')
ax0.set_ylabel('Membrane potential (mV)')
ax0.legend()

ax1.set_title('Excitatory postsynaptic currents')
ax1.plot(time, ampa1, c='black', label='slow rise')
ax1.plot(time, ampa2, c='crimson', label='instant rise')
ax1.set_xlabel('Time (ms)')
ax1.set_ylabel('Current (nS)')
ax1.legend()

fig.tight_layout()
b.show()
```



6.2 NMDA synapses

NMDA receptors, like AMPA receptors, are ionotropic glutamate receptors. They are responsible for excitatory synaptic transmission and have slower kinetics than AMPA receptors. NMDA receptors are voltage-dependent and require the postsynaptic membrane to be depolarized to remove the magnesium block and allow the influx of sodium and calcium ions.

In this example we show:

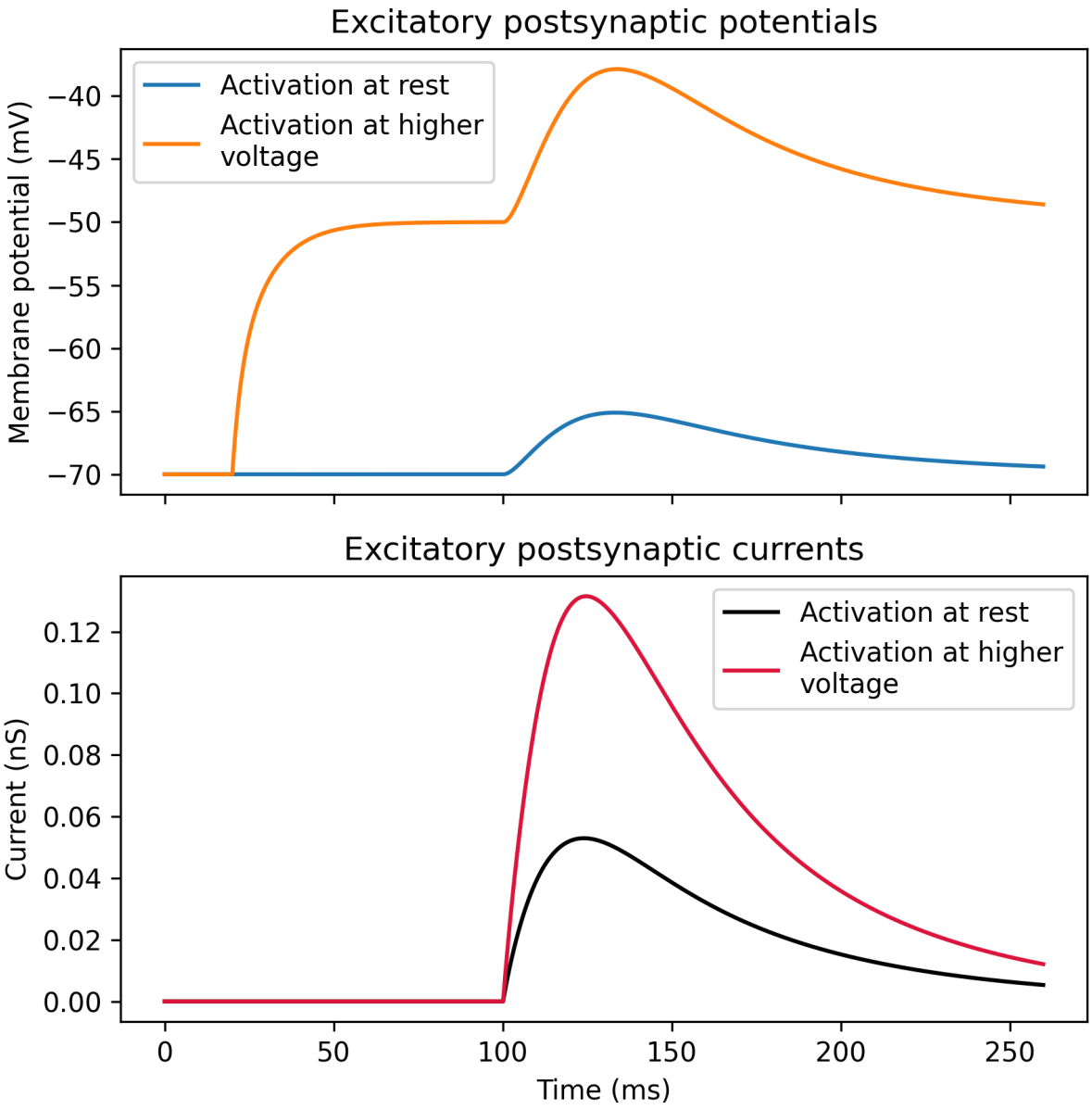
- How to add and activate NMDA synapses in a few-compartmental model.
- How NMDA synapses behave when activated at different membrane voltages.

```
import brian2 as b
from brian2.units import ms, mV, nS, pA, pF

from dendrify import Dendrite, NeuronModel, Soma

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create a simple 3-compartment neuron model
soma = Soma('soma', cm_abs=100*pF, gl_abs=10*nS)
dend = Dendrite('dend', cm_abs=50*pF, gl_abs=5*nS)
dend.synapse('NMDA', tag='foo', g=2*nS, t_rise=10*ms, t_decay=60*ms)

# Merge the compartments into a single neuron model
model = NeuronModel([(soma, dend, 10*nS)], v_rest=-70*mV)

# Create 2 neurons (no somatic spiking for simplicity)
neurons = model.make_neurongroup(2, method='euler')

# Each neuron receives a spike at t=10ms at different dendrites
input_spikes = b.SpikeGeneratorGroup(2, [0, 1], [100, 100]*ms)
slow_ampa = b.Synapses(input_spikes, neurons, on_pre='s_NMDA_foo_dend += 1')
slow_ampa.connect(i=0, j=0)
fast_ampa = b.Synapses(input_spikes, neurons, on_pre='s_NMDA_foo_dend += 1')
fast_ampa.connect(i=1, j=1)

# Create monitors
variables = ['V_dend', 'I_NMDA_foo_dend']
M = b.StateMonitor(neurons, variables, record=True)

# Run the simulation
b.run(20*ms)
neurons[1].I_ext_dend = 200*pA
b.run(240*ms)

# Visualize results
time = M.t/ms
vd1, vd2 = M.V_dend[0]/mV, M.V_dend[1]/mV
nmda1, nmda2 = M.I_NMDA_foo_dend[0]/nS, M.I_NMDA_foo_dend[1]/nS

fig, axes = b.subplots(2, 1, sharex=True, figsize=(6, 6))
ax0, ax1 = axes

ax0.set_title('Excitatory postsynaptic potentials')
ax0.plot(time, vd1, label='Activation at rest')
ax0.plot(time, vd2, label='Activation at higher\nvoltage')
ax0.set_ylabel('Membrane potential (mV)')
ax0.legend()

ax1.set_title('Excitatory postsynaptic currents')
ax1.plot(time, nmda1, c='black', label='Activation at rest')
ax1.plot(time, nmda2, c='crimson', label='Activation at higher\nvoltage')
ax1.set_xlabel('Time (ms)')
ax1.set_ylabel('Current (nS)')
```

(continues on next page)

(continued from previous page)

```
ax1.legend()

fig.tight_layout()
b.show()
```



6.3 GABA synapses

GABA receptors are the most common type of inhibitory ionotropic receptors in the brain. They have fast kinetics and are responsible for inhibitory synaptic transmission.

In this example we show:

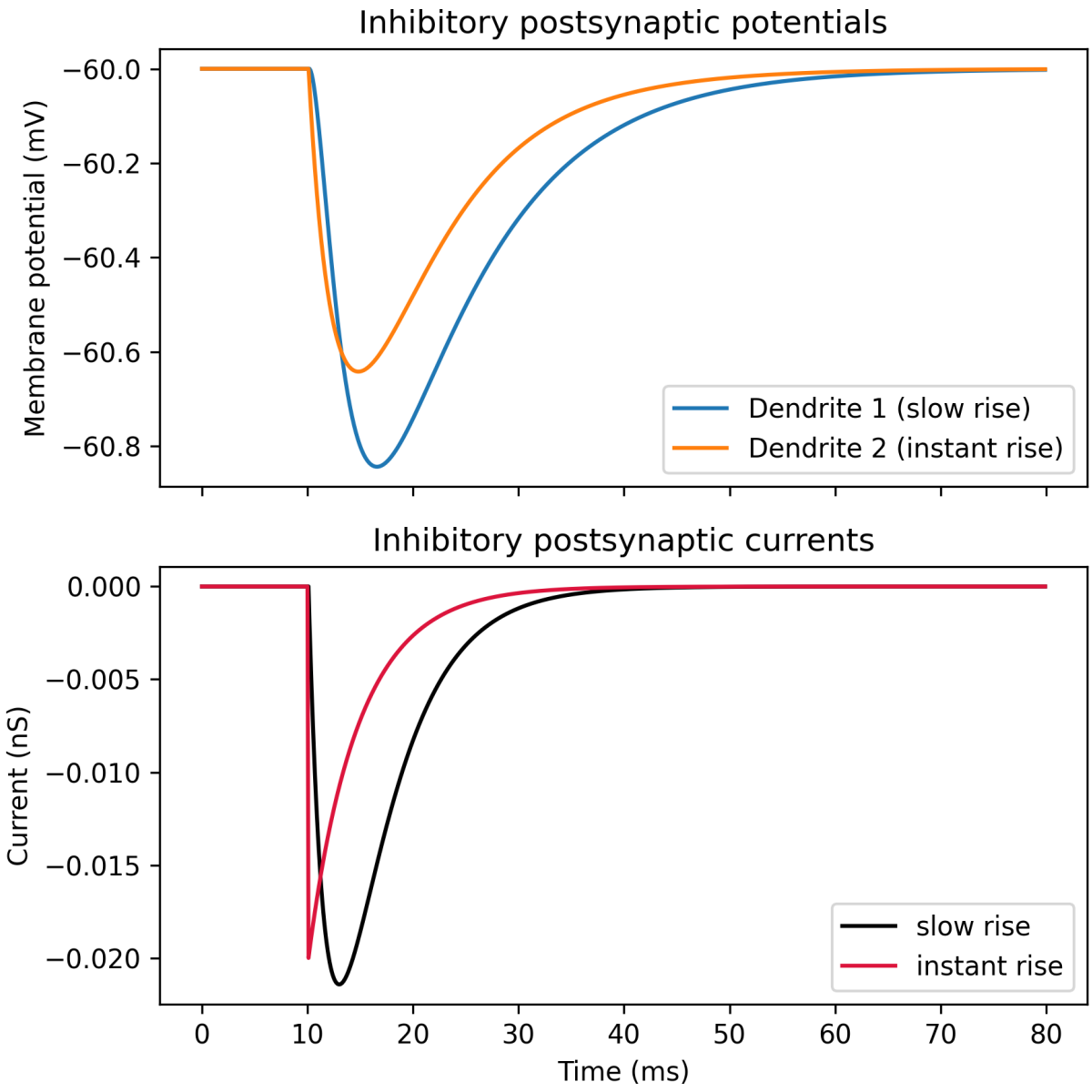
- How to add and activate GABA synapses at various model compartments.
- How to adjust the synaptic conductance and kinetics of GABA synapses.

```
import brian2 as b
from brian2.units import ms, mV, nS, pF

from dendrify import Dendrite, NeuronModel, Soma

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create a simple 3-compartment neuron model
soma = Soma('soma', cm_abs=100*pF, gl_abs=10*nS)
dend1 = Dendrite('dend1', cm_abs=50*pF, gl_abs=5*nS)
dend2 = Dendrite('dend2', cm_abs=50*pF, gl_abs=5*nS)

# Add GABA synapse with rise and decay kinetics to dend1
dend1.synapse('GABA', tag='slow', g=1*nS, t_rise=2*ms, t_decay=5*ms)

# Add GABA synapse with instantaneous rise and decay kinetics to dend2
dend2.synapse('GABA', tag='fast', g=1*nS,  t_decay=5*ms)

# Merge the compartments into a single neuron model
model = NeuronModel([(soma, dend1, 10*nS),
                     (dend1, dend2, 10*nS)],
                    v_rest=-60*mV)

# Create 2 neurons (no somatic spiking for simplicity)
neurons = model.make_neurongroup(2, method='euler')

# Each neuron receives a spike at t=10ms at different dendrites
input_spikes = b.SpikeGeneratorGroup(2, [0, 1], [10, 10]*ms)
slow_gaba = b.Synapses(input_spikes, neurons, on_pre='s_GABA_slow_dend1 += 1')
slow_gaba.connect(i=0, j=0)
fast_gaba = b.Synapses(input_spikes, neurons, on_pre='s_GABA_fast_dend2 += 1')
fast_gaba.connect(i=1, j=1)

# Create monitors
variables = ['V_dend1', 'V_dend2', 'I_GABA_slow_dend1', 'I_GABA_fast_dend2']
M = b.StateMonitor(neurons, variables, record=True)

# Run the simulation
b.run(80*ms)

# Visualize results
time = M.t/ms
vd1 = M.V_dend1[0]/mV
```

(continues on next page)

(continued from previous page)

```
vd2 = M.V_dend2[1]/mV
ampa1 = M.I_GABA_slow_dend1[0]/nS
ampa2 = M.I_GABA_fast_dend2[1]/nS

fig, axes = b.subplots(2, 1, sharex=True, figsize=(6, 6))
ax0, ax1 = axes

ax0.set_title('Inhibitory postsynaptic potentials')
ax0.plot(time, vd1, label='Dendrite 1 (slow rise)')
ax0.plot(time, vd2, label='Dendrite 2 (instant rise)')
ax0.set_ylabel('Membrane potential (mV)')
ax0.legend()

ax1.set_title('Inhibitory postsynaptic currents')
ax1.plot(time, ampa1, c='black', label='slow rise')
ax1.plot(time, ampa2, c='crimson', label='instant rise')
ax1.set_xlabel('Time (ms)')
ax1.set_ylabel('Current (nS)')
ax1.legend()

fig.tight_layout()
b.show()
```


Inhibitory postsynaptic potentials
Membrane potential (mV)
−60.0
−60.2
−60.4
−60.6
−60.8
Dendrite 1 (slow rise)
Dendrite 2 (instant rise)
Inhibitory postsynaptic currents
Current (nS)
0.000
−0.005
−0.010
−0.015
−0.020
slow rise
instant rise
0
10
20
30
40
50
60
70
80
Time (ms)

CHAPTER

SEVEN

VALIDATION TESTS

7.1 Input resistance

Input resistance (`Rin`) determines how much a neuron depolarizes in response to a steady current. It is a useful metric of a neuron's excitability; neurons with high `Rin` depolarize more in response to a given current than neurons with low `Rin`. `Rin` is often measured experimentally by injecting a small current `I` into the neuron and measuring the steady-state change in its membrane potential `V`. Using Ohm's law, `Rin` can be estimated as `Rin = V/I`.

In this example we show:

- How to calculate `Rin` in a point neuron model.
- How `Rin` is affected by changes in the neuron's membrane leak conductance `gl`.

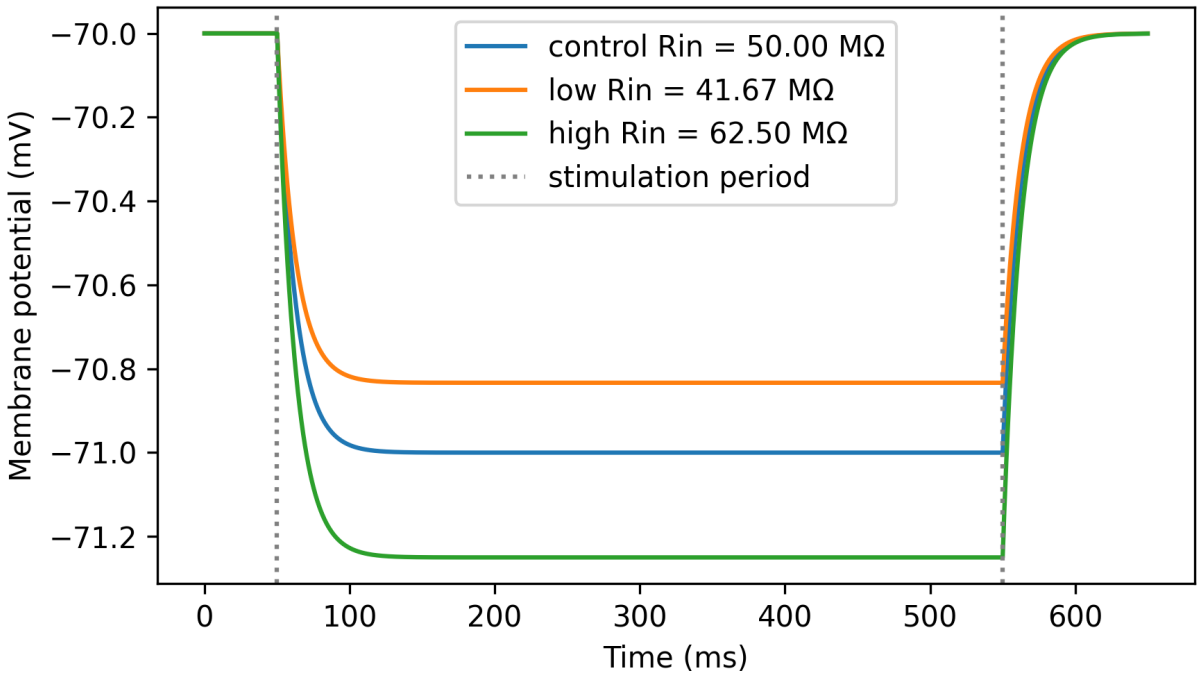
Note: We also scale the neuron's membrane capacitance `cm` to maintain a constant membrane time constant (`m = cm/gl`).

```
import brian2 as b
from brian2.units import Mohm, ms, mV, nS, pA, pF

from dendrify import PointNeuronModel

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Parameters
g_leakage = 20*nS  # membrane leak conductance
capacitance = 250*pF  # membrane capacitance
EL = -70*mV  # resting potential

# Create neuron models
control = PointNeuronModel(model='leakyIF', cm_abs=capacitance,
                           gl_abs=g_leakage, v_rest=EL)

low_rin = PointNeuronModel(model='leakyIF', cm_abs=capacitance*1.2,
                           gl_abs=g_leakage*1.2, v_rest=EL)

high_rin = PointNeuronModel(model='leakyIF', cm_abs=capacitance*0.8,
                            gl_abs=g_leakage*0.8, v_rest=EL)

# Create NeuronGroups (no threshold or reset conditions for simplicity)
control_neuron = control.make_neurongroup(N=1, method='euler')
low_rin_neuron = low_rin.make_neurongroup(N=1, method='euler')
```

(continues on next page)

(continued from previous page)

```
high_rin_neuron = high_rin.make_neurongroup(N=1, method='euler')

# Record voltages
control_monitor = b.StateMonitor(control_neuron, 'V', record=0)
low_rin_monitor = b.StateMonitor(low_rin_neuron, 'V', record=0)
high_rin_monitor = b.StateMonitor(high_rin_neuron, 'V', record=0)

# Run simulation
I = -20*pA  # current pulse amplitude
b.run(50*ms)
for n in [control_neuron, low_rin_neuron, high_rin_neuron]:
    n.I_ext = -20*pA
b.run(500*ms)
for n in [control_neuron, low_rin_neuron, high_rin_neuron]:
    n.I_ext = 0*pA
b.run(100*ms)

# Calculate Rin
Rin_control = (min(control_monitor.V[0]) - control_monitor.V[0][500]) / I
Rin_low = (min(low_rin_monitor.V[0]) - low_rin_monitor.V[0][500]) / I
Rin_high = (min(high_rin_monitor.V[0]) - high_rin_monitor.V[0][500]) / I

# Plot results
b.figure(figsize=(6, 3.5))
b.plot(control_monitor.t/ms, control_monitor.V[0]/mV,
       label='control Rin = {:.2f} M'.format(Rin_control / Mohm))
b.plot(low_rin_monitor.t/ms, low_rin_monitor.V[0]/mV,
       label='low Rin = {:.2f} M'.format(Rin_low / Mohm))
b.plot(high_rin_monitor.t/ms, high_rin_monitor.V[0]/mV,
       label='high Rin = {:.2f} M'.format(Rin_high / Mohm))
b.axvline(50, ls=':', c='gray', label='stimulation period')
b.axvline(550, ls=':', c='gray')
b.xlabel('Time (ms)')
b.ylabel('Membrane potential (mV)')
b.legend()
b.tight_layout()
b.show()
```



7.2 Frequency-current curve

A frequency-current curve (F-I curve) is the function that relates the net current `I` flowing into a neuron to its firing rate `F`.

In this example we show:

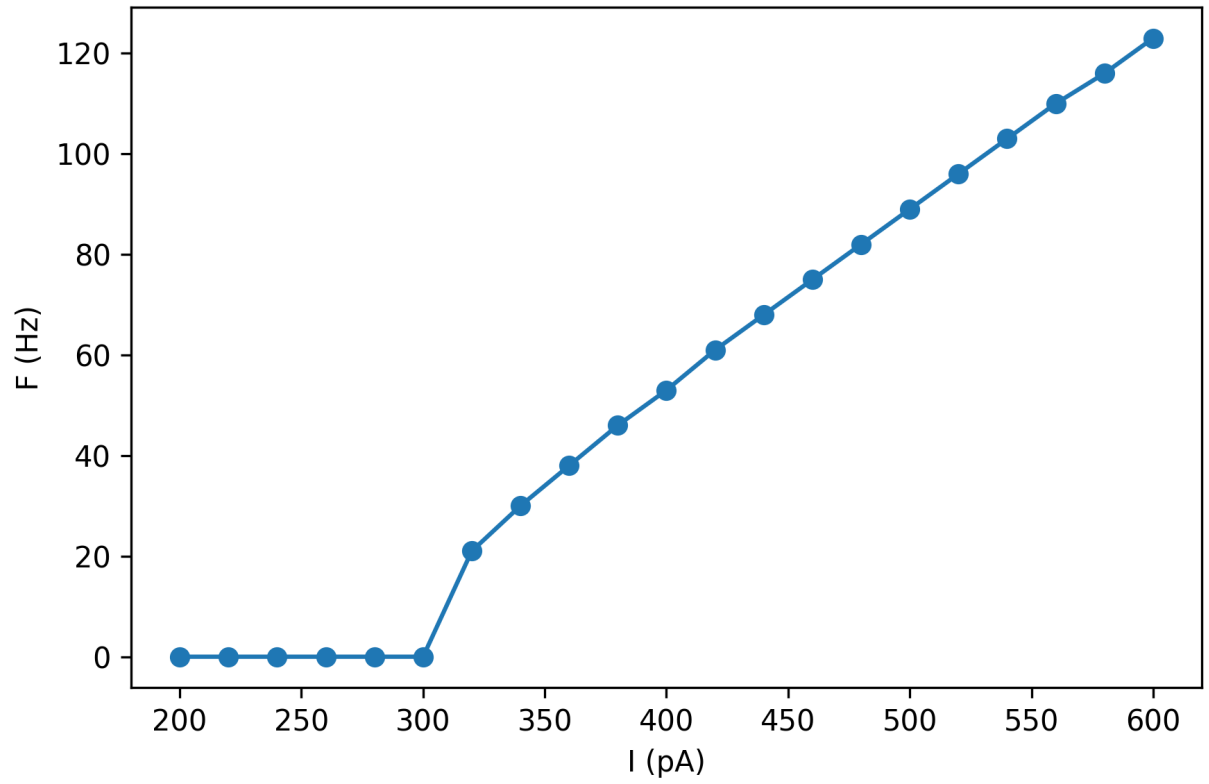
- How to calculate the somatic F-I curve for a simple 2-compartment neuron model.
- How to perform the above experiment in a vectorized and efficient manner.

```
import brian2 as b
from brian2.units import ms, mV, nS, pA, pF

from dendrify import Dendrite, NeuronModel, Soma

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create neuron model
soma = Soma('soma', cm_abs=200*pF, gl_abs=10*nS)
dend = Dendrite('dend', cm_abs=50*pF, gl_abs=2.5*nS)
model = NeuronModel([(soma, dend, 15*nS)], v_rest=-65*mV)

# Range of current amplitudes to test
I = range(200, 620, 20) * pA

# Create neuron group
"""Instead of creating a single neuron, we create a group of neurons, each with
a different value of ``I_ext``. This allows us to calculate the F-I curve in a
```

(continues on next page)

(continued from previous page)

```
single simulation."""
neurons = model.make_neurongroup(len(I), method='euler',
                                 threshold='V_soma > -40*mV',
                                 reset='V_soma = -55*mV',
                                 refractory=4*ms)

# Record spike times
spikes = b.SpikeMonitor(neurons)

# Run simulation
sim_time = 1000*ms
neurons.I_ext_soma = I
b.run(sim_time)

# Visualize F-I curve
F = [len(s) / sim_time for s in spikes.spike_trains().values()]
b.figure(figsize=(6, 4))
b.plot(I/pA, F, 'o-')
b.xlabel('I (pA)')
b.ylabel('F (Hz)')
b.tight_layout()
b.show()
```



7.3 Membrane time constant

In this example, we show how to calculate a neuron's membrane time constant `m`, a metric that describes how quickly the membrane potential `V` decays to its steady-state value after some perturbation. In simple RC circuits, `m` is calculated as the product of the membrane capacitance `C` and the membrane resistance `R`. However, in neurons, `m` is also affected by voltage-gated conductances or other non-linearities.

Experimentally, `m` is often calculated by fitting an exponential function to the membrane potential `V` trace after applying a small negative current pulse at rest.

Here we explore:

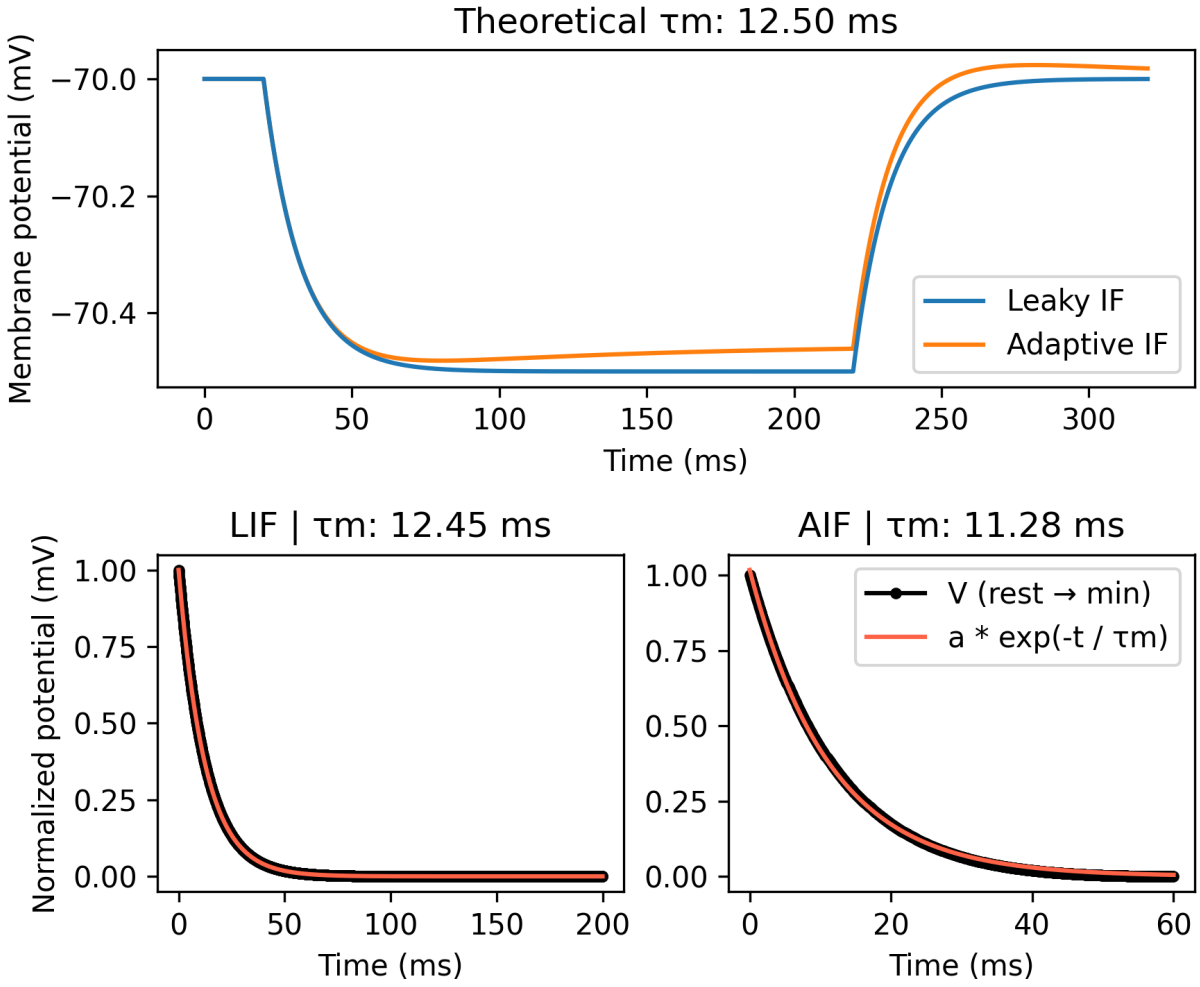
- How to calculate `m` for a neuron model experimentally.
- How `m` is affected by the presence of voltage-gated conductances, such as an adaptation current.

```
import brian2 as b
from brian2.units import ms, mV, nS, pA, pF
from scipy.optimize import curve_fit

from dendrify import PointNeuronModel

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create neuron models
GL = 20*nS  # membrane leak conductance
CM = 250*pF  # membrane capacitance
EL = -70*mV  # resting potential
tau_theory = CM / GL  # theoretical membrane time constant

lif = PointNeuronModel(model='leakyIF', cm_abs=CM, gl_abs=GL, v_rest=EL)
aif = PointNeuronModel(model='adaptiveIF', cm_abs=CM, gl_abs=GL, v_rest=EL)
aif.add_params({'tauw': 100*ms, 'a': 2*nS})

# Create NeuronGroups (no threshold or reset conditions for simplicity)
lif_neuron = lif.make_neurongroup(N=1, method='euler')
aif_neuron = aif.make_neurongroup(N=1, method='euler')

# Record voltages
lif_monitor = b.StateMonitor(lif_neuron, 'V', record=0)
aif_monitor = b.StateMonitor(aif_neuron, 'V', record=0)

# Run simulation
I = -10*pA  # current pulse amplitude
t0 = 20*ms  # time to start current pulse
t_stim = 200*ms  # duration of current pulse

b.run(t0)
lif_neuron.I_ext, aif_neuron.I_ext = I, I
b.run(t_stim)
lif_neuron.I_ext, aif_neuron.I_ext = 0*pA, 0*pA
b.run(100*ms)

# Analysis code
```

(continues on next page)

(continued from previous page)

```
def func(t, a, tau):
    """Exponential decay function"""
    return a * b.exp(-t / tau)


def get_tau(trace, t0):
    dt = b.defaultclock.dt
    Vmin = min(trace)
    time_to_peak = list(trace).index(Vmin)
    # Find voltage from current-start to min value
    voltages = trace[int(t0/dt): time_to_peak] / mV
    # Min-max normalize voltages
    v_norm = (voltages - voltages.min()) / (voltages.max() - voltages.min())
    # Fit exp decay function to normalized data
    X = b.arange(0, len(v_norm)) * dt / ms
    popt, _ = curve_fit(func, X, v_norm)
    return popt, X, v_norm


# Plot results
popt_lif, X_lif, v_norm_lif = get_tau(lif_monitor.V[0], t0)
popt_aif, X_aif, v_norm_aif = get_tau(aif_monitor.V[0], t0)

fig, axes = b.subplot_mosaic("""
                             AA
                             BC
                             """, layout='constrained', figsize=[6, 5])
ax0, ax1, ax2 = axes.values()
ax0.plot(lif_monitor.t/ms, lif_monitor.V[0]/mV, label='Leaky IF')
ax0.plot(aif_monitor.t/ms, aif_monitor.V[0]/mV, label='Adaptive IF', zorder=0)
ax0.set_title('Theoretical m: {:.2f} ms'.format(tau_theory/ms))
ax0.set_ylabel('Membrane potential (mV)')
ax0.legend()
ax1.plot(X_lif, v_norm_lif, 'ko-', ms=3)
ax1.plot(X_lif, func(X_lif, *popt_lif), c='tomato')
ax1.set_ylabel('Normalized potential (mV)')
ax1.set_title(f'LIF | m: {popt_lif[1]:.2f} ms')
ax2.plot(X_aif, v_norm_aif, 'ko-', label='V (rest \u2192 min)', ms=3)
ax2.plot(X_aif, func(X_aif, *popt_aif), label='a * exp(-t / m)', c='tomato')
ax2.set_title(f'AIF | m: {popt_aif[1]:.2f} ms')
ax2.legend()
for ax in axes.values():
    ax.set_xlabel('Time (ms)')
fig.tight_layout()
b.show()
```



7.4 Dendritic attenuation

The attenuation of currents traveling along the somatodendritic axis is an intrinsic property of biological neurons and is due to the morphology and cable properties of their dendritic trees. (also see Tran-van-Minh et al, 2015).

In this example, we show:

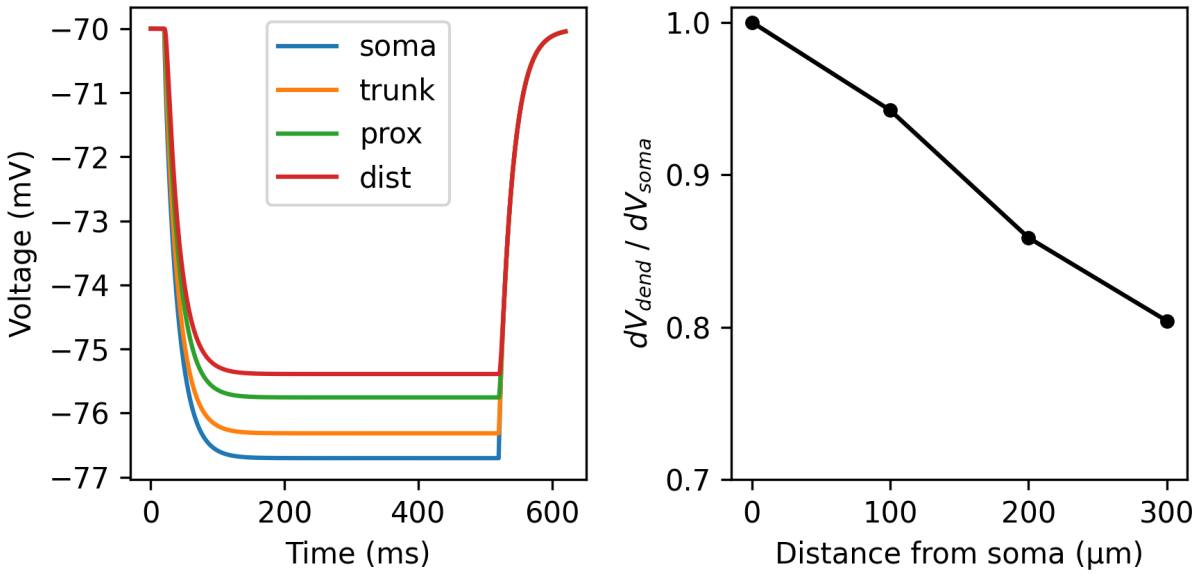
- How to measure the dendritic, distance-dependent voltage attenuation of a long current pulse injected at the soma.

```
import brian2 as b
from brian2.units import cm, ms, mV, ohm, pA, pF, uF, um, uS

from dendrify import Dendrite, NeuronModel, Soma

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create neuron model
soma = Soma('soma', length=25*um, diameter=25*um)
```

(continues on next page)

(continued from previous page)

```
trunk = Dendrite('trunk', length=100*um, diameter=1.5*um)
prox = Dendrite('prox', length=100*um, diameter=1.2*um)
dist = Dendrite('dist', length=100*um, diameter=1*um)

# Create a neuron group
connections = [(soma, trunk), (trunk, prox), (prox, dist)]
model = NeuronModel(connections, cm=1*uF/(cm**2), gl=50*uS/(cm**2),
                    v_rest=-70*mV, r_axial=400*ohm*cm)
neuron = model.make_neurongroup(1, method='euler')  # no spiking for simplicity

# Monitor voltages
M = b.StateMonitor(neuron, ['V_soma', 'V_trunk', 'V_prox', 'V_dist'],
                   record=True)

# Run simulation
b.run(20*ms)
neuron.I_ext_soma = -10*pA
b.run(500*ms)
neuron.I_ext_soma = 0*pA
b.run(100*ms)

# Analyse and plot results
time = M.t/ms
vs = M.V_soma[0]/mV
vt = M.V_trunk[0]/mV
vp = M.V_prox[0]/mV
vd = M.V_dist[0]/mV
voltages = [vs, vt, vp, vd]
delta_v = [min(v) - v[0] for v in voltages]
ratio = [i/delta_v[0] for i in delta_v]
distances = range(0, 400, 100)
names = ['soma', 'trunk', 'prox', 'dist']

fig, axes = b.subplots(1, 2, figsize=(6, 3))
ax0, ax1 = axes
for i, v in enumerate(voltages):
    ax0.plot(time, v, label=names[i])
ax0.set_ylabel('Voltage (mV)')
ax0.set_xlabel('Time (ms)')
ax0.legend()

ax1.plot(distances, ratio, 'ko-', ms=4)
ax1.set_ylabel(r'$dV_{dend}$ / $dV_{soma}$')
ax1.set_xlabel('Distance from soma (m)')
ax1.set_yticks(b.arange(.7, 1, .1))

fig.tight_layout()
b.show()
```



7.5 Dendritic I/O curve

Dendritic integration can be quantified by comparing the observed depolarization resulting from the quasi-simultaneous activation of the same synaptic inputs, and the arithmetic sum of individual EPSPs (expected membrane depolarization). The dendritic input-output (I/O) relationship is easily described by plotting observed vs. expected depolarizations for different numbers of co-activated synapses (also see Tran-van-Minh et al, 2015).

In this example, we show:

- How to calculate the dendritic I/O curve in a simple compartmental model.
- How active dendritic conductances affect the I/O curve.
- How to perform the above experiment in a vectorized and efficient manner.

```
import brian2 as b
from brian2.units import ms, mV, nS, pF

from dendrify import Dendrite, NeuronModel, Soma

b.prefs.codegen.target = 'numpy'  # faster for simple simulations

# Create neuron model
soma = Soma('soma', cm_abs=200*pF, gl_abs=10*nS)
dend = Dendrite('dend', cm_abs=50*pF, gl_abs=2.5*nS)
dend.dspikes('Na', g_rise=30*nS, g_fall=15*nS)
dend.synapse('AMPA', tag='x', g=3*nS,  t_decay=2*ms)
dend.synapse('NMDA', tag='x', g=3*nS,  t_decay=50*ms)

model = NeuronModel([(soma, dend, 15*nS)], v_rest=-65*mV)
model.config_dspikes('Na', threshold=-35*mV,
                     duration_rise=1.2*ms, duration_fall=2.4*ms,
                     offset_fall=0.2*ms, refractory=5*ms,
```

(continues on next page)

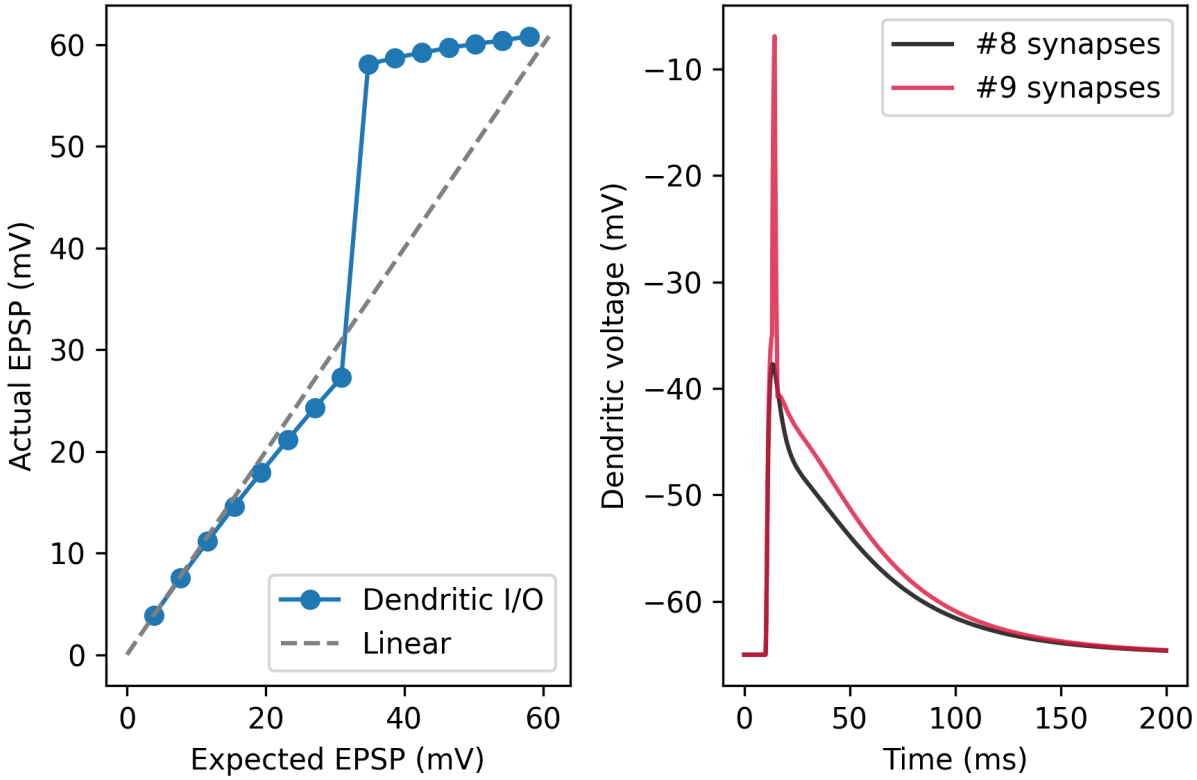
(continued from previous page)

```
                    reversal_rise='E_Na', reversal_fall='E_K')

# Create neuron group
"""Instead of creating a single neuron, we create a group of neurons, each
receiving a different number of synapses. This allows us to calculate the
dendritic I/O curve efficiently in a single simulation."""
N_syn = 15  # number of synapses
neurons = model.make_neurongroup(N_syn, method='euler',
                                 threshold='V_soma > -40*mV',
                                 reset='V_soma = -55*mV',
                                 refractory=4*ms)

# Create input source
start = 10*ms
isi = 0.1*ms  # inter-spike interval of input synapses
spiketimes = [(start + (i*isi)) for i in range(N_syn)]
I = b.SpikeGeneratorGroup(N_syn, range(N_syn), spiketimes)

# Connect input to neurons
synaptic_effect = "s_AMPA_x_dend += 1.0; s_NMDA_x_dend += 1.0"
S = b.Synapses(I, neurons, on_pre=synaptic_effect)
# 1st neuron receives 1 synapse, 2nd neuron receives 2 synapses, etc.
S.connect('j >= i')

# Record dendritic voltage
M = b.StateMonitor(neurons, ['V_dend'], record=True)

# Run simulation
b.run(200 * ms)

# Visualize results
time = M.t/ms
v = M.V_dend/mV
v_rest = v[0][0]
u_epsp = max(v[0]) - v_rest
expected = [u_epsp * (i+1) for i in range(N_syn)]
actual = [max(v[i]) - v_rest for i in range(N_syn)]
linear = b.linspace(0, max(actual))

fig, axes = b.subplots(1, 2, figsize=(6, 4))
ax0, ax1 = axes

ax0.plot(expected, actual, 'o-', label='Dendritic I/O')
ax0.plot(linear, linear, '--', color='gray', label='Linear')
ax0.set_xlabel('Expected EPSP (mV)')
ax0.set_ylabel('Actual EPSP (mV)')
ax0.legend()

ax1.plot(time, v[7], label='#8 synapses', c='black', alpha=0.8)
ax1.plot(time, v[8], label='#9 synapses', c='crimson', alpha=0.8)
ax1.set_xlabel('Time (ms)')
ax1.set_ylabel('Dendritic voltage (mV)')
```

(continues on next page)

(continued from previous page)

```
ax1.legend()
fig.tight_layout()
b.show()
```



CHAPTER

EIGHT

MODEL LIBRARY

8.1 Leaky membrane

$$C\frac{dV}{dt} = -g_L(V - E_L) + I$$

Symbol	Description
C	membrane capacitance
V	membrane potential
g_L	leakage conductance
E_L	leakage reversal potential
I	input current

8.2 Somatic spiking models

8.2.1 Leaky Integrate-and-Fire

$$C\frac{dV}{dt} = -g_L(V - E_L) + I$$

Spike mechanism:

$$\text{if } V \geq V_\theta \text{ then } V \to V_r$$

Symbol	Description
V_θ	spike threshold
V_r	reset potential

Examples:

- *Frequency-current curve*
- *LIF network + inhibition*

8.2.2 Adaptive Integrate-and-Fire

$$C\frac{dV}{dt} = -g_L(V - E_L) - w + I$$

$$\tau_w\frac{dw}{dt} = a(V - E_L) - w$$

Spike mechanism:

$$\text{if } V \geq V_\theta \text{ then } \begin{cases} V \to V_r \\ w \to w + b \end{cases}$$

Symbol	Description
w	adaptation current
a	maximal adaptation conductance
b	spike-triggered adaptation current
$_w$	adaptation time constant
V_θ	spike threshold
V_r	reset potential

Examples:

- *Other adaptive models*

8.2.3 Conductance-based Adaptive Integrate-and-Fire

$$C\frac{dV}{dt} = -g_L(V - E_L) - w + I$$

$$w = g_A(V - E_A)$$

$$\tau_A\frac{dg_A}{dt} = \bar{g_A}|V - E_A|^\gamma - g_A$$

Spike mechanism:

$$\text{if } V \geq V_\theta \text{ then } \begin{cases} V \to V_r \\ g_A \to g_A + \delta g_A \end{cases}$$

Symbol	Description
w	adaptation current
$\bar{g_A}$	maximal adaptation conductance
E_A	reversal potential of the adaptation
$_A$	adaptation time constant
δg_A	spike-triggered adaptation conductance
γ	steepness of the adaptation
V_θ	spike threshold
V_r	reset potential

Examples:

- *Other adaptive models*

8.2.4 Adaptive Exponential Integrate-and-Fire

$$C\frac{dV}{dt} = -g_L(V - E_L) + g_L\Delta_T \exp\left(\frac{V - V_T}{\Delta_T}\right) - w + I$$

$$\tau_w \frac{dw}{dt} = a(V - E_L) - w$$

Spike mechanism:

$$\text{if } V \geq V_\theta \text{ then } \begin{cases} V \to V_r \\ w \to w + b \end{cases}$$

Symbol	Description
w	adaptation current
a	maximal adaptation conductance
b	spike-triggered adaptation current
V_T	voltage threshold
Δ_T	slope factor
$_w$	adaptation time constant
V_θ	effective spike threshold
V_r	reset potential

Examples:

- *AdEx neuron*
- *AdEx network + synapses*

8.3 Synapses

8.3.1 AMPA

$$I_{\text{AMPA}} = \bar{g}_{\text{AMPA}}(E_{\text{AMPA}} - V)s(t)$$

$$\frac{ds}{dt} = \frac{-s}{\tau^{\text{decay}}_{\text{AMPA}}}$$

At presynaptic firing time:

$$s \to s + 1$$

Symbol	Description
$\bar{g}_{\text{AMPA}}$	maximal AMPA conductance
E_{AMPA}	AMPA reversal potential
$\tau^{\text{decay}}_{\text{AMPA}}$	AMPA decay time constant
s	channel state variable
V	membrane potential

Examples:

- *AMPA synapses*
- *AdEx network + synapses*

8.3.2 AMPA (rise & decay)

$$I_{\text{AMPA}} = \bar{g}_{\text{AMPA}}(E_{\text{AMPA}} - V)x(t)$$

$$\frac{dx}{dt} = \frac{-x}{\tau^{\text{decay}}_{\text{AMPA}}} + s(t)$$

$$\frac{ds}{dt} = \frac{-s}{\tau^{\text{rise}}_{\text{AMPA}}}$$

At presynaptic firing time:

$$s \rightarrow s + 1$$

Symbol	Description
$\bar{g}_{\text{AMPA}}$	maximal AMPA conductance
E_{AMPA}	AMPA reversal potential
$\tau^{\text{decay}}_{\text{AMPA}}$	AMPA decay time constant
s	rise state variable
x	decay state variable
V	membrane potential

Examples:

- *AMPA synapses*

8.3.3 NMDA

$$I_{\text{NMDA}} = \bar{g}_{\text{NMDA}}(E_{\text{NMDA}} - V)s(t)\sigma(V)$$

$$\frac{ds}{dt} = \frac{-s}{\tau_{\text{NMDA}}^{\text{decay}}}$$

$$\sigma(V) = \frac{1}{1 + \frac{\left[\text{Mg}^{2+}\right]_o}{\beta} \cdot \exp\left(-\alpha\left(V - \gamma\right)\right)}$$

At presynaptic firing time:

$$s \rightarrow s + 1$$

Symbol	Description
$\bar{g}_{\text{NMDA}}$	maximal NMDA conductance
E_{NMDA}	NMDA reversal potential
$\tau_{\text{NMDA}}^{\text{decay}}$	NMDA decay time constant
s	channel state variable
α	the steepness of Magnesium unblock
β	the sensitivity of Magnesium unblock
γ	offset of the Magnesium unblock
$[\text{Mg}^{2+}]_{\text{o}}$	external Magnesium concentration

Examples:

- *AdEx network + synapses*

8.3.4 NMDA (rise & decay)

$$I_{\text{NMDA}} = \bar{g}_{\text{NMDA}}(E_{\text{NMDA}} - V)x(t)\sigma(V)$$

$$\frac{dx}{dt} = \frac{-x}{\tau_{\text{NMDA}}^{\text{decay}}} + s(t)$$

$$\frac{ds}{dt} = \frac{-s}{\tau_{\text{NMDA}}^{\text{rise}}}$$

$$\sigma(V) = \frac{1}{1 + \frac{\left[\text{Mg}^{2+}\right]_o}{\beta} \cdot \exp\left(-\alpha\left(V - \gamma\right)\right)}$$

At presynaptic firing time:

$$s \rightarrow s + 1$$

Symbol	Description
$\bar{g}_{\text{NMDA}}$	maximal NMDA conductance
E_{NMDA}	NMDA reversal potential
$\tau^{\text{decay}}_{\text{NMDA}}$	NMDA decay time constant
s	rise state variable
x	decay state variable
α	the steepness of Magnesium unblock
β	the sensitivity of Magnesium unblock
γ	offset of the Magnesium unblock
$[\text{Mg}^{2+}]_{\text{o}}$	external Magnesium concentration

Examples:

- *NMDA synapses*

8.3.5 GABA

$$I_{\text{GABA}} = \bar{g}_{\text{GABA}}(E_{\text{GABA}} - V)s(t)$$

$$\frac{ds}{dt} = \frac{-s}{\tau^{\text{decay}}_{\text{GABA}}}$$

At presynaptic firing time:

$$s \to s + 1$$

Symbol	Description
$\bar{g}_{\text{GABA}}$	maximal GABA conductance
E_{GABA}	GABA reversal potential
$\tau^{\text{decay}}_{\text{GABA}}$	GABA decay time constant
s	channel state variable
V	membrane potential

Examples:

- *GABA synapses*
- *LIF network + inhibition*

8.3.6 GABA (rise & decay)

$$I_{\text{GABA}} = \bar{g}_{\text{GABA}}(E_{\text{GABA}} - V)x(t)$$

$$\frac{dx}{dt} = \frac{-x}{\tau_{\text{GABA}}^{\text{decay}}} + s(t)$$

$$\frac{ds}{dt} = \frac{-s}{\tau_{\text{GABA}}^{\text{rise}}}$$

At presynaptic firing time:

$$s \rightarrow s + 1$$

Symbol	Description
$\bar{g}_{\text{GABA}}$	maximal GABA conductance
E_{GABA}	GABA reversal potential
$\tau_{\text{GABA}}^{\text{decay}}$	GABA decay time constant
s	rise state variable
x	decay state variable
V	membrane potential

Examples:

- *NMDA synapses*

8.4 Study material

https://neuronaldynamics.epfl.ch/online/Ch1.html

https://neuronaldynamics.epfl.ch/online/Ch3.html

https://link.springer.com/chapter/10.1007/978-0-387-87708-2_2

https://link.springer.com/chapter/10.1007/978-0-387-87708-2_7#Sec1

CHAPTER

NINE

CLASSES

9.1 Soma

class dendrify.compartment.**Soma**(*name*, *model='leakyIF'*, *length=None*, *diameter=None*, *cm=None*, *gl=None*, *cm_abs=None*, *gl_abs=None*, *r_axial=None*, *v_rest=None*, *scale_factor=1.0*, *spine_factor=1.0*)

Bases: *Compartment*

A class representing a somatic compartment in a neuron model.

This class automatically generates and handles all differential equations and parameters needed to describe a somatic compartment and any currents (synaptic, dendritic, noise) passing through it.

See also:

Soma acts as a wrapper for Compartment with slight changes to account for certain somatic properties. For a full list of its methods and attributes, please see: *Compartment*.

Parameters

- **name** (*str*) – A unique name used to tag compartment-specific equations and parameters. It is also used to distinguish the various compartments belonging to the same *NeuronModel*.
- **model** (*str, optional*) – A keyword for accessing Dendrify's library models. Custom models can also be provided but they should be in the same formattable structure as the library models. Available options: `'leakyIF'` (default), `'adaptiveIF'`, `'adex'`.
- **length** (*Quantity, optional*) – A compartment's length.
- **diameter** (*Quantity, optional*) – A compartment's diameter.
- **cm** (*Quantity, optional*) – Specific capacitance (usually F / cm^2).
- **gl** (*Quantity, optional*) – Specific leakage conductance (usually S / cm^2).
- **cm_abs** (*Quantity, optional*) – Absolute capacitance (usually pF).
- **gl_abs** (*Quantity, optional*) – Absolute leakage conductance (usually nS).
- **r_axial** (*Quantity, optional*) – Axial resistance (usually Ohm * cm).
- **v_rest** (*Quantity, optional*) – Resting membrane voltage.
- **scale_factor** (*float, optional*) – A global area scale factor, by default `1.0`.
- **spine_factor** (*float, optional*) – A dendritic area scale factor to account for spines, by default `1.0`.

Examples

```
>>> # specifying equations only:
>>> compX = Soma('nameX', 'leakyIF')
>>> # specifying equations and ephys properties:
>>> compY = Soma('nameY', 'adaptiveIF', length=100*um, diameter=1*um,
>>>                     cm=1*uF/(cm**2), gl=50*uS/(cm**2))
>>> # specifying equations and absolute ephys properties:
>>> compY = Soma('nameZ', 'adaptiveIF', cm_abs=100*pF, gl_abs=20*nS)
```

9.2 Dendrite

class dendrify.compartment.**Dendrite**(*name*, *model='passive'*, *length=None*, *diameter=None*, *cm=None*, *gl=None*, *cm_abs=None*, *gl_abs=None*, *r_axial=None*, *v_rest=None*, *scale_factor=1.0*, *spine_factor=1.0*)

Bases: *Compartment*

A class that automatically generates and handles all differential equations and parameters needed to describe a dendritic compartment, its active mechanisms, and any currents (synaptic, dendritic, ionic, noise) passing through it.

See also:

Dendrite inherits all the methods and attributes of its parent class *Compartment*. For a complete list, please refer to the documentation of the latter.

Parameters

- **name** (*str*) – A unique name used to tag compartment-specific equations and parameters. It is also used to distinguish the various compartments belonging to the same *NeuronModel*.
- **model** (*str, optional*) – A keyword for accessing Dendrify's library models. Dendritic compartments are by default set to `'passive'`.
- **length** (*Quantity, optional*) – A compartment's length.
- **diameter** (*Quantity, optional*) – A compartment's diameter.
- **cm** (*Quantity, optional*) – Specific capacitance (usually F / cm^2).
- **gl** (*Quantity, optional*) – Specific leakage conductance (usually S / cm^2).
- **cm_abs** (*Quantity, optional*) – Absolute capacitance (usually pF).
- **gl_abs** (*Quantity, optional*) – Absolute leakage conductance (usually nS).
- **r_axial** (*Quantity, optional*) – Axial resistance (usually Ohm * cm).
- **v_rest** (*Quantity, optional*) – Resting membrane voltage.
- **scale_factor** (*float, optional*) – A global area scale factor, by default `1.0`.
- **spine_factor** (*float, optional*) – A dendritic area scale factor to account for spines, by default `1.0`.

Examples

```
>>> # specifying equations only:
>>> compX = Dendrite('nameX')
>>> # specifying equations and ephys properties:
>>> compY = Dendrite('nameY', length=100*um, diameter=1*um,
>>>                     cm=1*uF/(cm**2), gl=50*uS/(cm**2))
>>> # specifying equations and absolute ephys properties:
>>> compY = Dendrite('nameZ', cm_abs=100*pF, gl_abs=20*nS)
```

Methods:

dspikes	Adds the ionic mechanisms and parameters needed for dendritic spiking.

Attributes:

event_names	Returns a list of all dSpike event names created for a single dendrite.
events	Returns a dictionary of all dSpike events created for a single dendrite.
parameters	Returns a dictionary of all parameters that have been generated for a single compartment.

dspikes(*name*, *threshold=None*, *g_rise=None*, *g_fall=None*, *duration_rise=None*, *duration_fall=None*, *reversal_rise=None*, *reversal_fall=None*, *offset_fall=None*, *refractory=None*)

Adds the ionic mechanisms and parameters needed for dendritic spiking. Under the hood, this method creates the equations, conditions and actions to take advantage of Brian's custom events. dSpikes are generated through the sequential activation of a positive (sodium or calcium-like) and a negative current (potassium-like current) when a specified dSpike threshold is crossed.

Hint: The dendritic spiking mechanism as implemented here has three distinct phases.

INACTIVE PHASE:

When the dendritic voltage is subthreshold OR the simulation step is within the refractory period. dSpikes cannot be generated during this phase.

RISE PHASE:

When the dendritic voltage crosses the dSpike threshold AND the refractory period has elapsed. This triggers the instant activation of a positive current that is deactivated after a specified amount of time (`duration_rise`). Also a new refractory period begins.

FALL PHASE:

This phase starts automatically with a delay (`offset_fall`) after the dSpike threshold is crossed. A negative current is activated instantly and then is deactivated after a specified amount of time (`duration_fall`).

Parameters

- **name** (*str*) – A unique name to describe a single dSpike type.

- **threshold** (*Quantity, optional*) – The membrane voltage threshold for dendritic spiking.
- **g_rise** (*Quantity, optional*) – The max conductance of the channel that is activated during the rise (depolarization phase).
- **g_fall** (*Quantity, optional*) – The max conductance of the channel that is activated during the fall (repolarization phase).
- **duration_rise** (*Quantity, optional*) – The duration of g_rise staying open.
- **duration_fall** (*Quantity, optional*) – The duration of g_fall staying open.
- **reversal_rise** (*(Quantity, str), optional*) – The reversal potential of the channel that is activated during the rise (depolarization) phase.
- **reversal_fall** (*(Quantity, str), optional*) – The reversal potential of the channel that is activated during the fall (repolarization) phase.
- **offset_fall** (*Quantity, optional*) – The delay for the activation of g_rise.
- **refractory** (*Quantity, optional*) – The time interval required before dSpike can be activated again.

property event_names

Returns a list of all dSpike event names created for a single dendrite.

Return type
list

property events

Returns a dictionary of all dSpike events created for a single dendrite.

Returns
Keys: event names, values: events conditions.

Return type
dict

property parameters

Returns a dictionary of all parameters that have been generated for a single compartment.

Return type
dict

9.3 NeuronModel

class dendrify.neuronmodel.**NeuronModel**(*connections*, *cm=None*, *gl=None*, *r_axial=None*, *v_rest=None*, *scale_factor=None*, *spine_factor=None*)

Bases: object

Creates a multicompartmental neuron model by connecting individual compartments and merging their equations, parameters and custom events.This model can then be used for creating a population of neurons through Brian's NeuronGroup. This class also contains useful methods for managing model properties and for automating the initialization of custom events and simulation parameters.

Tip: Dendrify aims to facilitate the development of reduced, **few-compartmental** I&F models that help us study how key dendritic properties may affect network-level functions. It is not designed to substitute morphologically

and biophysically detailed neuron models, commonly used for highly-accurate, single-cell simulations. If you are interested in the latter category of models, please see Brian's SpatialNeuron.

Parameters

- **connections** (*list[tuple[Compartment, Compartment, str | Quantity]]*) – A description of how the various compartments belonging to the same neuron model should be connected.
- **cm** (*Quantity, optional*) – Specific capacitance (usually F / cm^2).
- **gl** (*Quantity, optional*) – Specific leakage conductance (usually S / cm^2).
- **r_axial** (*Quantity, optional*) – Axial resistance (usually Ohm * cm).
- **v_rest** (*Quantity, optional*) – Resting membrane voltage.
- **scale_factor** (*float, optional*) – A global area scale factor, by default `1.0`.
- **spine_factor** (*float, optional*) – A dendritic area scale factor to account for spines, by default `1.0`.

Warning: Parameters set here affect all model compartments and can override any compartment-specific parameters.

Example

```
>>> # Valid format: [*(x, y, z)], where
>>> # x -> Soma or Dendrite object
>>> # y -> Soma or Dendrite object other than x
>>> # z -> 'half_cylinders' or 'cylinder_ + name' or brian2.nS unit
>>> #      (by default 'half_cylinders')
>>> soma = Soma(...)
>>> prox = Dendrite(...)
>>> dist = Dendrite(...)
>>> connections = [(soma, prox, 15*nS), (prox, dist, 10*nS)]
>>> model = NeuronModel(connections)
```

Methods:

add_equations	Allows adding custom equations.
add_params	Allows specifying extra/custom parameters.
as_graph	Plots a graph-like representation of a NeuronModel using the Graph class and the Fruchterman-Reingold force-directed algorithm from Networkx.
config_dspikes	Configure the parameters for dendritic spiking.
make_neurongroup	Returns a Brian2 NeuronGroup object from a NeuronModel.

Attributes:

equations	Returns a string containing all model equations.
event_actions	Returns a dictionary containing all event actions for dendritic spiking.
event_names	Returns a list of all event names for dendritic spiking.
events	Returns a dictionary containing all model custom events for dendritic spiking.
parameters	Returns a dictionary containing all model parameters.

add_equations(*eqs*)

Allows adding custom equations.

Parameters
eqs (*str*) – A string of Brian-compatible equations.

add_params(*params_dict*)

Allows specifying extra/custom parameters.

Parameters
params_dict (*dict*) – A dictionary of parameters.

as_graph(*figsize=[6, 4]*, *fontsize=10*, *fontcolor='white'*, *scale_nodes=1*, *color_soma='#4C6C92'*, *color_dendrites='#A7361C'*, *alpha=1*, *scale_edges=1*, *seed=None*)

Plots a graph-like representation of a NeuronModel using the Graph class and the Fruchterman-Reingold force-directed algorithm from Networkx.

Parameters

- **fontsize** (*int, optional*) – The size in pt of each node's name, by default `10`.
- **fontcolor** (*str, optional*) – The color of each node's name, by default `'white'`.
- **scale_nodes** (*float, optional*) – Percentage change in node size, by default `1`.
- **color_soma** (*str, optional*) – Somatic node color, by default `'#4C6C92'`.
- **color_dendrites** (*str, optional*) – Dendritic nodes color, by default `'#A7361C'`.
- **alpha** (*float, optional*) – Nodes color opacity, by default `1`.
- **scale_edges** (*float, optional*) – The percentage change in edges length, by default `1`.
- **seed** (*int, optional*) – Set the random state for deterministic node layouts, by default `None`. .

config_dspikes(*event_name*, *threshold=None*, *duration_rise=None*, *duration_fall=None*, *reversal_rise=None*, *reversal_fall=None*, *offset_fall=None*, *refractory=None*)

Configure the parameters for dendritic spiking.

Parameters

- **event_name** (*str*) – A unique name referring to a specific dSpike type.
- **threshold** (*Quantity, optional*) – The membrane voltage threshold for dendritic spiking.
- **duration_rise** (*Quantity, optional*) – The duration of g_rise staying open.
- **duration_fall** (*Quantity, optional*) – The duration of g_fall staying open.

- **reversal_rise** (*((Quantity, str), optional*) – The reversal potential of the channel that is activated during the rise (depolarization) phase.
- **reversal_fall** (*((Quantity, str), optional*) – The reversal potential of the channel that is activated during the fall (repolarization) phase.
- **offset_fall** (*Quantity, optional*) – The delay for the activation of g_rise.
- **refractory** (*Quantity, optional*) – The time interval required before dSpike can be activated again.

property equations

Returns a string containing all model equations.

Returns
All model equations.

Return type
str

property event_actions

Returns a dictionary containing all event actions for dendritic spiking.

Returns
All event actions for dendritic spiking

Return type
list

property event_names

Returns a list of all event names for dendritic spiking.

Returns
All event names for dendritic spiking

Return type
list

property events

Returns a dictionary containing all model custom events for dendritic spiking.

Returns
All model custom events for dendritic spiking.

Return type
dict

make_neurongroup(*N*, *method='euler'*, *threshold=None*, *reset=None*, *second_reset=None*, *spike_width=None*, *refractory=False*, *init_rest=True*, *init_events=True*, *show=False*, ***kwargs*)

Returns a Brian2 NeuronGroup object from a NeuronModel. If a second reset is provided, it also returns a Synapses object to implement somatic action potentials with a more realistic shape which also unlocks dendritic backpropagation. This method can also take all parameters that are accepted by Brian's NeuronGroup.

Parameters

- **N** (*int*) – The number of neurons in the group.
- **method** (*str, optional*) – The numerical integration method. Either a string with the name of a registered method (e.g. "euler") or a function that receives an *Equations* object and returns the corresponding abstract code, by default `'euler'`.

- **threshold** (*str, optional*) – The condition which produces spikes. Should be a single line boolean expression.
- **reset** (*str, optional*) – The (possibly multi-line) string with the code to execute on reset.
- **refractory** (*(Quantity, str), optional*) – Either the length of the refractory period (e.g. `2*ms`), a string expression that evaluates to the length of the refractory period after each spike (e.g. `'(1 + rand())*ms'`), or a string expression evaluating to a boolean value, given the condition under which the neuron stays refractory after a spike (e.g. `'v > -20*mV'`).
- **second_reset** (*str, optional*) – Option to include a second reset for more realistic somatic spikes.
- **spike_width** (*Quantity, optional*) – The time interval between the two resets.
- **init_rest** (*bool, optional*) – Option to automatically initialize the voltages of all compartments at the specified resting potentials, by default True.
- **init_events** (*bool, optional*) – Option to automatically initialize all custom events that required for dendritic spiking, by default True.
- **show** (*bool, optional*) – Option to print the automatically initialized parameters, by default False.
- ****kwargs** (*optional*) – All other parameters accepted by Brian's NeuronGroup.

Returns
If no second reset is added, it returns a NeuronGroup object. Otherwise, it returns a tuple of (NeuronGroup, Synapses) objects.

Return type
Union[NeuronGroup, Tuple]

property parameters

Returns a dictionary containing all model parameters.

Returns
All model parameters.

Return type
dict

9.4 PointNeuronModel

class dendrify.neuronmodel.**PointNeuronModel**(*model='leakyIF'*, *length=None*, *diameter=None*, *cm=None*, *gl=None*, *cm_abs=None*, *gl_abs=None*, *v_rest=None*)

Bases: `object`

Like a *NeuronModel* but for point-neuron (single-compartment) models.

Parameters

- **model** (*str, optional*) – A keyword for accessing Dendrify's library models. Custom models can also be provided but they should be in the same formattable structure as the library models. Available options: `'leakyIF'` (default), `'adaptiveIF'`, `'adex'`.
- **length** (*Quantity, optional*) – The point neuron's length.
- **diameter** (*Quantity, optional*) – The point neuron's diameter.

- **cm** (*Quantity, optional*) – Specific capacitance (usually F / cm^2).
- **gl** (*Quantity, optional*) – Specific leakage conductance (usually S / cm^2).
- **cm_abs** (*Quantity, optional*) – Absolute capacitance (usually pF).
- **gl_abs** (*Quantity, optional*) – Absolute leakage conductance (usually nS).
- **v_rest** (*Quantity, optional*) – Resting membrane voltage.

Methods:

add_equations	Allows adding custom equations.
add_params	Allows specifying extra/custom parameters.
make_neurongroup	Create a NeuronGroup object with the specified number of neurons.
noise	Adds a stochastic noise current.
synapse	Adds synaptic currents equations and parameters.

Attributes:

area	Returns a compartment's surface area (open cylinder) based on its length and diameter.
capacitance	Returns a compartment's absolute capacitance.
dimensionless	Checks if a compartment has been flagged as dimensionless.
equations	Returns all differential equations that describe a single compartment and the mechanisms that have been added to it.
g_leakage	A compartment's absolute leakage conductance.
parameters	Returns all the parameters that have been generated for a single compartment.

add_equations(*eqs*)

Allows adding custom equations.

Parameters
eqs (*str*) – A string of Brian-compatible equations.

add_params(*params_dict*)

Allows specifying extra/custom parameters.

Parameters
params_dict (*dict*) – A dictionary of parameters.

property area

Returns a compartment's surface area (open cylinder) based on its length and diameter.

Return type
Quantity

property capacitance

Returns a compartment's absolute capacitance.

Return type
Quantity

property dimensionless

Checks if a compartment has been flagged as dimensionless.

Return type
bool

property equations

Returns all differential equations that describe a single compartment and the mechanisms that have been added to it.

Return type
str

property g_leakage

A compartment's absolute leakage conductance.

Return type
Quantity

make_neurongroup(*N*, ***kwargs*)

Create a NeuronGroup object with the specified number of neurons.

Parameters

- **N** (*int*) – The number of neurons in the group.
- ****kwargs** – Additional keyword arguments to be passed to the NeuronGroup constructor.

Returns
The created NeuronGroup object.

Return type
NeuronGroup

noise(*tau=20. * msecond*, *sigma=1. * pamp*, *mean=0. * amp*)

Adds a stochastic noise current. For more information see the Noise section: of Models and neuron groups

Parameters

- **tau** (Quantity, optional) – Time constant of the Gaussian noise, by default `20*ms`
- **sigma** (Quantity, optional) – Standard deviation of the Gaussian noise, by default `3*pA`
- **mean** (Quantity, optional) – Mean of the Gaussian noise, by default `0*pA`

property parameters

Returns all the parameters that have been generated for a single compartment.

Return type
dict

synapse(*channel*, *tag*, *g=None*, *t_rise=None*, *t_decay=None*, *scale_g=False*)

Adds synaptic currents equations and parameters. When only the decay time constant `t_decay` is provided, the synaptic model assumes an instantaneous rise of the synaptic conductance followed by an exponential decay. When both the rise `t_rise` and decay `t_decay` constants are provided, synapses are modelled as a sum of two exponentials. For more information see: Modeling Synapses by Arnd Roth & Mark C. W. van Rossum

Parameters

- **channel** (*str*) – Synaptic channel type. Available options: `'AMPA'`, `'NMDA'`, `'GABA'`.
- **tag** (*str*) – A unique name to distinguish synapses of the same type.

- **g** (Quantity) – Maximum synaptic conductance
- **t_rise** (Quantity) – Rise time constant
- **t_decay** (Quantity) – Decay time constant
- **scale_g** (*bool, optional*) – Option to add a normalization factor to scale the maximum conductance at 1 when synapses are modelled as a difference of exponentials (have both rise and decay kinetics), by default `False`.

Examples

```
>>> neuron = PointNeuronModel(...)
>>> # adding an AMPA synapse with instant rise & exponential decay:
>>> neuron.synapse('AMPA', tag='X', g=1*nS, t_decay=5*ms)
>>> # same channel, different conductance & source:
>>> neuron.synapse('AMPA', tag='Y', g=2*nS, t_decay=5*ms)
>>> # different channel with both rise & decay kinetics:
>>> neuron.synapse('NMDA', tag='X' g=1*nS, t_rise=5*ms, t_decay=50*ms)
```

9.5 Compartment

class dendrify.compartment.**Compartment**(*name*, *model='passive'*, *length=None*, *diameter=None*, *cm=None*, *gl=None*, *cm_abs=None*, *gl_abs=None*, *r_axial=None*, *v_rest=None*, *scale_factor=1.0*, *spine_factor=1.0*)

Bases: `object`

A class that automatically generates and handles all differential equations and parameters needed to describe a single compartment and any currents (synaptic, dendritic, noise) passing through it.

Parameters

- **name** (*str*) – A unique name used to tag compartment-specific equations and parameters. It is also used to distinguish the various compartments belonging to the same *NeuronModel*.
- **model** (*str, optional*) – A keyword for accessing Dendrify's library models. Custom models can also be provided but they should be in the same formattable structure as the library models. Available options: `'passive'` (default), `'adaptiveIF'`, `'leakyIF'`, `'adex'`.
- **length** (*Quantity, optional*) – A compartment's length.
- **diameter** (*Quantity, optional*) – A compartment's diameter.
- **cm** (*Quantity, optional*) – Specific capacitance (usually F / cm^2).
- **gl** (*Quantity, optional*) – Specific leakage conductance (usually S / cm^2).
- **cm_abs** (*Quantity, optional*) – Absolute capacitance (usually pF).
- **gl_abs** (*Quantity, optional*) – Absolute leakage conductance (usually nS).
- **r_axial** (*Quantity, optional*) – Axial resistance (usually Ohm * cm).
- **v_rest** (*Quantity, optional*) – Resting membrane voltage.
- **scale_factor** (*float, optional*) – A global area scale factor, by default `1.0`.
- **spine_factor** (*float, optional*) – A dendritic area scale factor to account for spines, by default `1.0`.

Examples

```
>>> # specifying equations only:
>>> compX = Compartment('nameX', 'leakyIF')
>>> # specifying equations and ephys properties:
>>> compY = Compartment('nameY', 'adaptiveIF', length=100*um, diameter=1*um,
>>>                     cm=1*uF/(cm**2), gl=50*uS/(cm**2))
>>> # specifying equations and absolute ephys properties:
>>> compY = Compartment('nameZ', 'adaptiveIF', cm_abs=100*pF, gl_abs=20*nS)
```

Attributes:

area	Returns a compartment's surface area (open cylinder) based on its length and diameter.
capacitance	Returns a compartment's absolute capacitance.
dimensionless	Checks if a compartment has been flagged as dimensionless.
equations	Returns all differential equations that describe a single compartment and the mechanisms that have been added to it.
g_leakage	A compartment's absolute leakage conductance.
parameters	Returns all the parameters that have been generated for a single compartment.

Methods:

connect	Connects two compartments (electrical coupling).
g_norm_factor	Calculates the normalization factor for synaptic conductance with t_rise and t_decay kinetics.
noise	Adds a stochastic noise current.
synapse	Adds synaptic currents equations and parameters.

property area

Returns a compartment's surface area (open cylinder) based on its length and diameter.

Return type
Quantity

property capacitance

Returns a compartment's absolute capacitance.

Return type
Quantity

connect(*other*, *g='half_cylinders'*)

Connects two compartments (electrical coupling).

Parameters

- **other** (Compartment) – Another compartment.
- **g** (str or Quantity, optional) – The coupling conductance. It can be set explicitly or calculated automatically (provided all necessary parameters exist). Available options: 'half_cylinders' (default), 'cylinder_<compartment name>'.

Warning: The automatic approaches require that both compartments to be connected have specified **length**, **diameter** and **axial resistance**.

Examples

```
>>> compX, compY = Compartment('x', **kwargs), Compartment('y', **kwargs)
>>> # explicit approach:
>>> compX.connect(compY, g=10*nS)
>>> # half cylinders (default):
>>> compX.connect(compY)
>>> # cylinder of one compartment:
>>> compX.connect(compY, g='cylinder_x')
```

property dimensionless

Checks if a compartment has been flagged as dimensionless.

Return type
bool

property equations

Returns all differential equations that describe a single compartment and the mechanisms that have been added to it.

Return type
str

property g_leakage

A compartment's absolute leakage conductance.

Return type
Quantity

static g_norm_factor(*t_rise*, *t_decay*)

Calculates the normalization factor for synaptic conductance with t_rise and t_decay kinetics.

Parameters: t_rise (Quantity): The rise time of the function. t_decay (Quantity): The decay time of the function.

Returns: float: The normalization factor for the g function.

noise(*tau=20. * msecond*, *sigma=1. * pamp*, *mean=0. * amp*)

Adds a stochastic noise current. For more information see the Noise section: of Models and neuron groups

Parameters

- **tau** (Quantity, optional) – Time constant of the Gaussian noise, by default `20*ms`
- **sigma** (Quantity, optional) – Standard deviation of the Gaussian noise, by default `3*pA`
- **mean** (Quantity, optional) – Mean of the Gaussian noise, by default `0*pA`

property parameters

Returns all the parameters that have been generated for a single compartment.

Return type
dict

synapse(*channel*, *tag*, *g=None*, *t_rise=None*, *t_decay=None*, *scale_g=False*)

Adds synaptic currents equations and parameters. When only the decay time constant `t_decay` is provided, the synaptic model assumes an instantaneous rise of the synaptic conductance followed by an exponential decay. When both the rise `t_rise` and decay `t_decay` constants are provided, synapses are modelled as a sum of two exponentials. For more information see: Modeling Synapses by Arnd Roth & Mark C. W. van Rossum

Parameters

- **channel** (*str*) – Synaptic channel type. Available options: `'AMPA'`, `'NMDA'`, `'GABA'`.
- **tag** (*str*) – A unique name to distinguish synapses of the same type.
- **g** (Quantity) – Maximum synaptic conductance
- **t_rise** (Quantity) – Rise time constant
- **t_decay** (Quantity) – Decay time constant
- **scale_g** (*bool, optional*) – Option to add a normalization factor to scale the maximum conductance at 1 when synapses are modelled as a difference of exponentials (have both rise and decay kinetics), by default `False`.

Examples

```
>>> comp = Compartment('comp')
>>> # adding an AMPA synapse with instant rise & exponential decay:
>>> comp.synapse('AMPA', tag='X', g=1*nS, t_decay=5*ms)
>>> # same channel, different conductance & source:
>>> comp.synapse('AMPA', tag='Y', g=2*nS, t_decay=5*ms)
>>> # different channel with both rise & decay kinetics:
>>> comp.synapse('NMDA', tag='X' g=1*nS, t_rise=5*ms, t_decay=50*ms)
```

9.6 EphysProperties

class dendrify.ephysproperties.**EphysProperties**(*name=None*, *length=None*, *diameter=None*, *cm=None*, *gl=None*, *cm_abs=None*, *gl_abs=None*, *r_axial=None*, *v_rest=None*, *scale_factor=1.0*, *spine_factor=1.0*)

Bases: `object`

A class for calculating various important electrophysiological properties for a single compartment.

Note: An EphysProperties object is automatically created and linked to a single compartment during the initialization of the latter.

Parameters

- **name** (*str, optional*) – A compartment's name.
- **length** (*Quantity, optional*) – A compartment's length.
- **diameter** (*Quantity, optional*) – A compartment's diameter.
- **cm** (*Quantity, optional*) – Specific capacitance (usually F / cm^2).

- **gl** (*Quantity, optional*) – Specific leakage conductance (usually S / cm^2).
- **cm_abs** (*Quantity, optional*) – Absolute capacitance (usually pF).
- **gl_abs** (*Quantity, optional*) – Absolute leakage conductance (usually nS).
- **r_axial** (*Quantity, optional*) – Axial resistance (usually Ohm * cm).
- **v_rest** (*Quantity, optional*) – Resting membrane voltage.
- **scale_factor** (*float, optional*) – A global area scale factor, by default `1.0`.
- **spine_factor** (*float, optional*) – A dendritic area scale factor to account for spines, by default `1.0`.

Attributes:

area	Returns compartment's surface area (open cylinder) based on its length and diameter.
capacitance	Returns a compartment's capacitance based on its specific capacitance (cm) and surface area.
g_cylinder	The conductance (of coupling currents) passing through a cylindrical compartment based on its dimensions and its axial resistance.
g_leakage	Returns a compartment's absolute leakage conductance based on its specific leakage conductance (gl) and surface area.
parameters	Returns a dictionary of all the major electrophysiological parameters that describe a single compartment.

Methods:

g_couple	The conductance (of coupling currents) between the centers of two adjacent cylindrical compartments, based on their dimensions and the axial resistance.

property area

Returns compartment's surface area (open cylinder) based on its length and diameter.

Returns
A compartment's surface area

Return type
Quantity

property capacitance

Returns a compartment's capacitance based on its specific capacitance (cm) and surface area. If an absolute capacitance (cm_abs) has been provided by the user, it returns this value instead.

Return type
Quantity

static g_couple(*comp1*, *comp2*)

The conductance (of coupling currents) between the centers of two adjacent cylindrical compartments, based on their dimensions and the axial resistance.

Parameters

- **comp1** (EphysProperties) – An EphysProperties object
- **comp2** (EphysProperties) – An EphysProperties object

Return type
Quantity

property g_cylinder

The conductance (of coupling currents) passing through a cylindrical compartment based on its dimensions and its axial resistance. To be used when then the total number of compartments is low and the adjacent-to-soma compartments are highly coupled with the soma.

Return type
Quantity

property g_leakage

Returns a compartment's absolute leakage conductance based on its specific leakage conductance (gl) and surface area. If an absolute leakage conductance (gl_abs) has been provided by the user, it returns this value instead.

Return type
Quantity

property parameters

Returns a dictionary of all the major electrophysiological parameters that describe a single compartment.

Return type
dict

CHAPTER

TEN

INDEX

CHAPTER

ELEVEN

SUPPORT

Dendrify was created by Michalis Pagkalos and is currently maintained by Michalis Pagkalos and Spyros Chavlis, both currently members of the Poirazi Lab .

If you run into any issues or have questions about using Dendrify, you can either:

- E-mail us at dendrify@dendrites.gr
- Open a new issue on GitHub

Tip: To facilitate communication, please include a minimal working example that reproduces your issue. This will help us understand your problem and provide a faster solution.

CHAPTER

TWELVE

RELEASE NOTES

12.1 Version 2.1.2

- Fixed a bug that could cause the wrong estimation of a somatic compartment's surface area.
- Other minor improvements.

12.2 Version 2.1.1

- A minor release to fix some bugs introduced by VS code auto-formatting.

12.3 Version 2.1.0

- Changed the order of dSpikes thresholding to be more compatible with other Brian 2 objects and increase overall simulation stability.
- Added the cadIF model as an option to PointNeuronModel.
- Completely redesigned the Library section of the documentation and added more mathematical descriptions for the all built-in Dendrify models.
- Added many new code examples.
- Minor improvements in the source code for better readability and maintainability.

12.4 Version 2.0.1

- Added Integrate-and-Fire with conductance based adaptation (cadIF) model.

12.5 Version 2.0.0

- New and improved implementation of dendritic spikes.
- New PointNeuronModel class for creating point-neuron models.
- New way for specifying the electrophysiological properties of neurons.
- Significantly improved error catching and exception handling.
- Fixed compatibility issues with Jupyter notebooks.
- More stable and robust code overall.
- Added tutorials and code examples.
- Improved documentation page.
- Added a support e-mail address.
- Many minor improvements, bug fixes and quality of life improvements.
- New logo.

Special thanks to Marcel Stimberg, Spyros Chavlis, Nikos Malakasis, Christos Karageorgiou Kaneen and Elisavet Kapetanou for their valuable feedback and suggestions for improving Dendrify.

12.6 Version 1.0.9

- Minor improvements.

12.7 Version 1.0.8

- Improved documentation.
- Minor improvements.

12.8 Version 1.0.5

- Improved documentation.
- Minor bug fixes.

12.9 Version 1.0.4

- Redesigned documentation page.
- Added more type hints.
- Improved compatibility with older Python versions.
- Minor bug fixes.

CHAPTER

THIRTEEN

IMPORTANT LITERATURE

Introducing the Dendrify framework for incorporating dendrites to spiking neural networks M Pagkalos, S Chavlis, P Poirazi DOI: https://doi.org/10.1038/s41467-022-35747-8

Brian 2, an intuitive and efficient neural simulator M Stimberg, R Brette, D FM Goodman DOI: https://doi.org/10.7554/eLife.47314

Contribution of sublinear and supralinear dendritic integration to neuronal computations A Tran-Van-Minh, R D Cazé, T Abrahamsson, L Cathala, B Gutkin, D A DiGregorio DOI: https://doi.org/10.3389/fncel.2015.00067

Pyramidal neurons: dendritic structure and synaptic integration N Spruston DOI: https://doi.org/10.1038/nrn2286

Reduced compartmental models of neocortical pyramidal cells P C Bush, T J Sejnowski DOI : https://doi.org/10.1016/0165-0270(93)90151-g

Book | Mathematical Foundations of Neuroscience (chapters 1, 2 & 7) G B Ermentrout, D H Terman Publisher's website: https://link.springer.com/book/10.1007/978-0-387-87708-2

Brian2CUDA: flexible and efficient simulation of spiking neural network models on GPUs D Alevi, M Stimberg, H Sprekeler, K Obermayer, M Augustin DOI: https://doi.org/10.3389/fninf.2022.883700

CHAPTER

FOURTEEN

CODE OF CONDUCT

14.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to make participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

14.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

14.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

14.4 Scope

This Code of Conduct applies within all project spaces, and it also applies when an individual is representing the project or its community in public spaces. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

14.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at dendrify@dendrites.gr. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

14.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

INDEX

N

P

S