

---

# **eFEL Documentation**

*Release 5.2.9+5.gada9b43*

**BBP, EPFL**

**Dec 08, 2023**



# CONTENTS

<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	Requirements . . . . .	3
1.2	Installation using pip . . . . .	3
1.3	Installing the C++ standalone library . . . . .	3
<b>2</b>	<b>Examples</b>	<b>5</b>
2.1	Python . . . . .	5
2.1.1	Quick start . . . . .	5
2.1.2	DEAP optimisation . . . . .	7
2.1.3	Reading different file formats . . . . .	13
<b>3</b>	<b>eFeature descriptions</b>	<b>15</b>
3.1	Implemented eFeatures (to be continued) . . . . .	15
3.1.1	Spike event features . . . . .	15
3.1.2	Spike shape features . . . . .	26
3.1.3	Voltage features . . . . .	43
3.2	Requested eFeatures . . . . .	51
3.2.1	C++ features . . . . .	51
3.2.2	Python features . . . . .	53
<b>4</b>	<b>Python API</b>	<b>57</b>
<b>5</b>	<b>Developer's Guide</b>	<b>61</b>
5.1	Requirements . . . . .	61
5.2	Forking and cloning the git repository . . . . .	62
5.3	Makefile . . . . .	62
5.4	Adding a new eFeature . . . . .	62
5.4.1	Picking a name . . . . .	62
5.4.2	Creating a branch . . . . .	63
5.4.3	Implementation . . . . .	63
5.4.4	Updating relevant files . . . . .	63
5.4.5	Adding a test . . . . .	63
5.4.6	Add documentation . . . . .	64
5.4.7	Pull request . . . . .	64
<b>6</b>	<b>Indices and tables</b>	<b>65</b>
	<b>Python Module Index</b>	<b>67</b>
	<b>Index</b>	<b>69</b>



The Electrophys Feature Extract Library (eFEL) allows neuroscientists to automatically extract eFeatures from time series data recorded from neurons (both in vitro and in silico). Examples are the action potential width and amplitude in voltage traces recorded during whole-cell patch clamp experiments. The user of the library provides a set of traces and selects the eFeatures to be calculated. The library will then extract the requested eFeatures and return the values to the user.

The core of the library is written in C++, and a Python wrapper is included. At the moment we provide a way to automatically compile and install the library as a Python module. Soon instructions will be added on how to link C++ code directly with the eFEL.

The source code of the eFEL is located on github: [BlueBrain/eFEL](https://github.com/BlueBrain/eFEL)



## INSTALLATION

### 1.1 Requirements

- Python
- Pip (installed by default in newer versions of Python)
- Numpy (will be installed automatically by pip)
- The instruction below are written assuming you have access to a command shell on Linux / UNIX / MacOSX / Cygwin

### 1.2 Installation using pip

The easiest way to install eFEL is to use `pip`:

```
pip install efel
```

In case you don't have administrator access this command might fail with a permission error. In that case you could install eFEL in your home directory:

```
pip install efel --user
```

Or you could use a [python virtual environment](#):

```
virtualenv pythonenv  
. ./pythonenv/bin/activate  
pip install git+git://github.com/BlueBrain/eFEL
```

### 1.3 Installing the C++ standalone library

If your system doesn't have it, install [CMake](#).

Make a new build directory:

```
mkdir build_cmake
```

Configure the build, replace `YOURINSTALLDIR` with the directory in which you want to install the efel library (e.g. `/usr/local`):

```
cd build_cmake
cmake .. -DCMAKE_INSTALL_PREFIX=YOURINSTALLDIR
```

Run the compilation and installation:

```
make install
```

This will have installed a static and shared library as:

```
YOURINSTALLDIR/lib/libefel.a
YOURINSTALLDIR/lib/libefel.so
```



## EXAMPLES

### 2.1 Python

#### 2.1.1 Quick start

First you need to import the module:

```
import efel
```

To get a list with all the available eFeature names:

```
efel.getFeatureNames()
```

The python function to extract eFeatures is `getFeatureValues(...)`. Below is a short example on how to use this function. The code and example trace are available [here](#):

```
"""Basic example 1 for eFEL"""

import efel
import numpy

def main():
    """Main"""

    # Use numpy to read the trace data from the txt file
    data = numpy.loadtxt('example_trace1.txt')

    # Time is the first column
    time = data[:, 0]
    # Voltage is the second column
    voltage = data[:, 1]

    # Now we will construct the datastructure that will be passed to eFEL

    # A 'trace' is a dictionary
    trace1 = {}

    # Set the 'T' (=time) key of the trace
    trace1['T'] = time
```

(continues on next page)

(continued from previous page)

```

# Set the 'V' (=voltage) key of the trace
trace1['V'] = voltage

# Set the 'stim_start' (time at which a stimulus starts, in ms)
# key of the trace
# Warning: this need to be a list (with one element)
trace1['stim_start'] = [700]

# Set the 'stim_end' (time at which a stimulus end) key of the trace
# Warning: this need to be a list (with one element)
trace1['stim_end'] = [2700]

# Multiple traces can be passed to the eFEL at the same time, so the
# argument should be a list
traces = [trace1]

# Now we pass 'traces' to the efel and ask it to calculate the feature
# values
traces_results = efel.getFeatureValues(traces,
                                       ['AP_amplitude', 'voltage_base'])

# The return value is a list of trace_results, every trace_results
# corresponds to one trace in the 'traces' list above (in same order)
for trace_results in traces_results:
    # trace_result is a dictionary, with as keys the requested eFeatures
    for feature_name, feature_values in trace_results.items():
        print("Feature %s has the following values: %s" % \
              (feature_name, ', '.join([str(x) for x in feature_values])))

if __name__ == '__main__':
    main()

```

The output of this example is:

```

Feature AP_amplitude has the following values: 72.5782441262, 46.3672552618, 41.
↪1546679158, 39.7631750953, 36.1614653031, 37.8489295737
Feature voltage_base has the following values: -75.446665721

```

This means that the eFEL found 5 action potentials in the voltage trace. The amplitudes of these APs are the result of the 'AP\_amplitude' feature.

The voltage before the start of the stimulus is measured by 'voltage\_base'.

Results are in mV.

## 2.1.2 DEAP optimisation

### Contents

- *DEAP optimisation*
  - *Introduction*
  - *Evaluation function*
  - *Setting up the algorithm*
  - *Running the code*

### Introduction

Using the eFEL, pyNeuron and the DEAP optimisation library one can very easily set up a genetic algorithm to fit parameters of a neuron model.

We propose this setup because it leverages the power of the Python language to load several software tools in a compact script. The DEAP (Distributed Evolutionary Algorithms in Python) allows you to easily switch algorithms. Parallelising your evaluation function over cluster computers becomes a matter of only adding a couple of lines to your code, thanks to `pyScoop`.

In this example we will assume you have installed `eFEL`, `pyNeuron` and `DEAP`

The code of the example below can be downloaded from [here](#)

To keep the example simple, let's start from a passive single compartmental model. The parameters to fit will be the conductance and reversal potential of the leak channel. We will simulate the model for 1000 ms, and at 500 ms a step current of 1.0 nA is injected until the end of the simulation.

The objective values of the optimisation will be the voltage before the current injection (i.e. the 'voltage\_base' feature), and the steady state voltage during the current injection at the end of the simulation ('steady\_state\_voltage').

### Evaluation function

We now have to use pyNeuron to define the evaluation function to be optimised. The input arguments are the parameters:

```
g_pas, e_pas
```

and the return values:

```
abs(voltage_base - target_voltage1)
abs(steady_state_voltage - target_voltage2)
```

This translates into the following file (let's call it 'deap\_efel\_eval1.py'):

```
import neuron
neuron.h.load_file('stdrun.hoc')

import efel

# pylint: disable=W0212
```

(continues on next page)

```
def evaluate(individual, target_voltage1=-80, target_voltage2=-60):
    """
    Evaluate a neuron model with parameters e_pas and g_pas, extracts
    eFeatures from resulting traces and returns a tuple with
    abs(voltage_base-target_voltage1) and
    abs(steady_state_voltage-target_voltage2)
    """

    neuron.h.v_init = target_voltage1

    soma = neuron.h.Section()

    soma.insert('pas')

    soma.g_pas = individual[0]
    soma.e_pas = individual[1]

    clamp = neuron.h.IClamp(0.5, sec=soma)

    stim_start = 500
    stim_end = 1000

    clamp.amp = 1.0
    clamp.delay = stim_start
    clamp.dur = 100000

    voltage = neuron.h.Vector()
    voltage.record(soma(0.5)._ref_v)

    time = neuron.h.Vector()
    time.record(neuron.h._ref_t)

    neuron.h.tstop = stim_end
    neuron.h.run()

    trace = {}
    trace['T'] = time
    trace['V'] = voltage
    trace['stim_start'] = [stim_start]
    trace['stim_end'] = [stim_end]
    traces = [trace]

    features = efel.getFeatureValues(traces, ["voltage_base",
                                             "steady_state_voltage"])
    voltage_base = features[0]["voltage_base"][0]
    steady_state_voltage = features[0]["steady_state_voltage"][0]

    return abs(target_voltage1 - voltage_base), \
           abs(target_voltage2 - steady_state_voltage)
```

## Setting up the algorithm

Now that we have an evaluation function we just have to pass this to the DEAP optimisation library. DEAP allows you to easily set up a genetic algorithm to optimise your evaluation function. Let us first import all the necessary components:

```
import random
import numpy

import deap
import deap.gp
import deap.benchmarks
from deap import base
from deap import creator
from deap import tools
from deap import algorithms
random.seed(1)
```

Next we define a number of constants that will be used as settings for DEAP later:

```
# Population size
POP_SIZE = 100
# Number of offspring in every generation
OFFSPRING_SIZE = 100

# Number of generations
NGEN = 300

# The parent and offspring population size are set the same
MU = OFFSPRING_SIZE
LAMBDA = OFFSPRING_SIZE
# Crossover probability
CXPB = 0.7
# Mutation probability, should sum to one together with CXPB
MUTPB = 0.3

# Eta parameter of cx and mut operators
ETA = 10.0
```

We have two parameters with the following bounds:

```
# The size of the individual is 2 (parameters g_pas and e_pas)
IND_SIZE = 2

LOWER = [1e-8, -100.0]
UPPER = [1e-4, -20.0]
```

As evolutionary algorithm we choose NSGA2:

```
SELECTOR = "NSGA2"
```

Let's create the DEAP individual and fitness. We set the weights of the fitness values to -1.0 so that the fitness function will be minimised instead of maximised:

```
creator.create("Fitness", base.Fitness, weights=[-1.0] * 2)
```

The individual will just be a list (of two parameters):

```
creator.create("Individual", list, fitness=creator.Fitness)
```

We want to start with individuals for which the parameters are picked from a uniform random distribution. Let's create a function that returns such a random list based on the bounds and the dimensions of the problem:

```
def uniform(lower_list, upper_list, dimensions):
    """Fill array """

    if hasattr(lower_list, '__iter__'):
        return [random.uniform(lower, upper) for lower, upper in
                zip(lower_list, upper_list)]
    else:
        return [random.uniform(lower_list, upper_list)
                for _ in range(dimensions)]
```

DEAP works with the concept of 'toolboxes'. The user defines genetic algorithm's individuals, operators, etc by registering them in a toolbox.

We first create the toolbox:

```
toolbox = base.Toolbox()
```

Then we register the 'uniform' function we defined above:

```
toolbox.register("uniformparams", uniform, LOWER, UPPER, IND_SIZE)
```

The three last parameters of this register call will be passed on to the 'uniform' function call

Now we can also register an individual:

```
toolbox.register(
    "Individual",
    tools.initIterate,
    creator.Individual,
    toolbox.uniformparams)
```

And a population as list of individuals:

```
toolbox.register("population", tools.initRepeat, list, toolbox.Individual)
```

The function to evaluate we defined above. Assuming you saved that files as 'deap\_efel\_eval1.py', we can import it as a module, and register the function:

```
import deap_efel_eval1
toolbox.register("evaluate", deap_efel_eval1.evaluate)
```

For the mutation and crossover operator we use builtin operators that are typically used with NSGA2:

```
toolbox.register(
    "mate",
    deap.tools.cxSimulatedBinaryBounded,
```

(continues on next page)

(continued from previous page)

```

eta=ETA,
low=LOWER,
up=UPPER)
toolbox.register("mutate", deap.tools.mutPolynomialBounded, eta=ETA,
                 low=LOWER, up=UPPER, indpb=0.1)

```

And then we specify the selector to be used:

```

toolbox.register(
    "select",
    tools.selNSGA2)

```

We initialise the population with the size of the offspring:

```

pop = toolbox.population(n=MU)

```

And register some statistics we want to print during the run of the algorithm:

```

first_stats = tools.Statistics(key=lambda ind: ind.fitness.values[0])
second_stats = tools.Statistics(key=lambda ind: ind.fitness.values[1])
stats = tools.MultiStatistics(obj1=first_stats, obj2=second_stats)
stats.register("min", numpy.min, axis=0)

```

The only thing that is left now is to run the algorithm in ‘main’:

```

if __name__ == '__main__':
    pop, logbook = algorithms.eaMuPlusLambda(
        pop,
        toolbox,
        MU,
        LAMBDA,
        CXPB,
        MUTPB,
        NGEN,
        stats,
        halloffame=None)

```

For your convenience the full code is in a code block below. It should be saved as ‘deap\_efel.py’.

## Running the code

Assuming that the necessary dependencies are installed correctly the optimisation can then be run with:

```

python deap_efel.py

```

The full code of ‘deap\_efel.py’:

```

import random
import numpy

import deap
import deap.gp

```

(continues on next page)

(continued from previous page)

```

import deap.benchmarks
from deap import base
from deap import creator
from deap import tools
from deap import algorithms

random.seed(1)
POP_SIZE = 100
OFFSPRING_SIZE = 100

NGEN = 300
ALPHA = POP_SIZE
MU = OFFSPRING_SIZE
LAMBDA = OFFSPRING_SIZE
CXPB = 0.7
MUTPB = 0.3
ETA = 10.0

SELECTOR = "NSGA2"

IND_SIZE = 2
LOWER = [1e-8, -100.0]
UPPER = [1e-4, -20.0]

creator.create("Fitness", base.Fitness, weights=[-1.0] * 2)
creator.create("Individual", list, fitness=creator.Fitness)

def uniform(lower_list, upper_list, dimensions):
    """Fill array """

    if hasattr(lower_list, '__iter__'):
        return [random.uniform(lower, upper) for lower, upper in
                zip(lower_list, upper_list)]
    else:
        return [random.uniform(lower_list, upper_list)
                for _ in range(dimensions)]

toolbox = base.Toolbox()
toolbox.register("uniformparams", uniform, LOWER, UPPER, IND_SIZE)
toolbox.register(
    "Individual",
    tools.initIterate,
    creator.Individual,
    toolbox.uniformparams)
toolbox.register("population", tools.initRepeat, list, toolbox.Individual)

import deap_efel_eval1
toolbox.register("evaluate", deap_efel_eval1.evaluate)

toolbox.register(

```

(continues on next page)



(continued from previous page)

```

"mate",
deap.tools.cxSimulatedBinaryBounded,
eta=ETA,
low=LOWER,
up=UPPER)
toolbox.register("mutate", deap.tools.mutPolynomialBounded, eta=ETA,
                 low=LOWER, up=UPPER, indpb=0.1)

toolbox.register("variate", deap.algorithms.varAnd)

toolbox.register(
    "select",
    tools.selNSGA2)

pop = toolbox.population(n=MU)

first_stats = tools.Statistics(key=lambda ind: ind.fitness.values[0])
second_stats = tools.Statistics(key=lambda ind: ind.fitness.values[1])
stats = tools.MultiStatistics(obj1=first_stats, obj2=second_stats)
stats.register("min", numpy.min, axis=0)

if __name__ == '__main__':
    pop, logbook = algorithms.eaMuPlusLambda(
        pop,
        toolbox,
        MU,
        LAMBDA,
        CXPB,
        MUTPB,
        NGEN,
        stats,
        halloffame=None)

```

### 2.1.3 Reading different file formats

Neo is a Python package which provides support for reading a wide range of neurophysiology file formats, including Spike2, NeuroExplorer, AlphaOmega, Axon, Blackrock, Plexon and Tdt.

The function `efel.io.load_neo_file()` reads data from any of the file formats supported by Neo and formats it for use in eFEL.

As an example, suppose we have an `.abf` file containing a single trace. Since eFEL requires information about the start and end times of the current injection stimulus, we provide these times as well as the filename:

```

import efel

data = efel.io.load_neo_file("path/first_file.abf", stim_start=200, stim_end=700)

```

Since some file formats can contain multiple recording episodes (e.g. trials) and multiple signals per episode, the function returns traces in a list of lists, like this:

```
data : [Segment_1, Segment_2, ..., Segment_n]
      with Segment_1 = [Trace_1, Trace_2, ..., Trace_n]
```

Since our file contains only a single recording episode, our list of traces is:

```
traces = data[0]
```

which we pass to eFEL as follows:

```
features = efel.getFeatureValues(traces, ['AP_amplitude', 'voltage_base'])
```

### Stimulus information within the file

Some file formats can store information about the current injection stimulus. In this second example, the file contains an Epoch object named “stimulation”, so we don’t need to explicitly specify *stim\_start* and *stim\_end*:

```
data2 = efel.io.load_neo_file("path/second_file.h5")
```

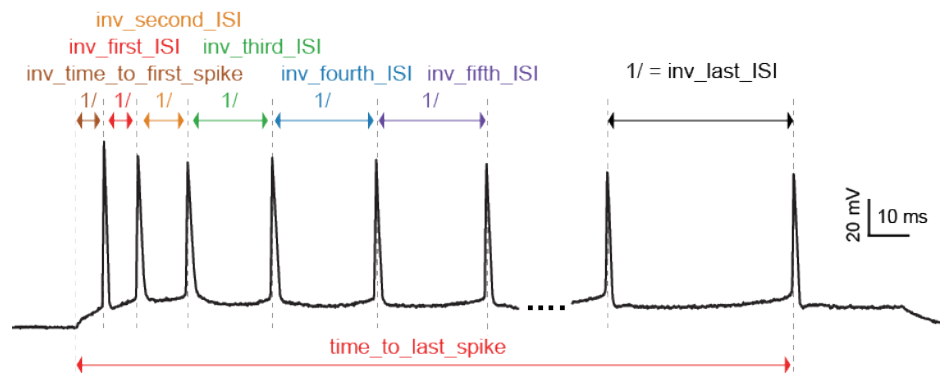
## EFEATURE DESCRIPTIONS

A pdf document describing the eFeatures is available [here](#).

Not every eFeature has a description in this document yet, the complete set will be available shortly.

### 3.1 Implemented eFeatures (to be continued)

#### 3.1.1 Spike event features



#### LibV1 : time\_to\_first\_spike

Time from the start of the stimulus to the maximum of the first peak

- **Required features:** peak\_time
- **Units:** ms
- **Pseudocode:**

```
time_to_first_spike = peakttime[0] - stimstart
```

### LibV5 : time\_to\_second\_spike

Time from the start of the stimulus to the maximum of the second peak

- **Required features:** peak\_time
- **Units:** ms
- **Pseudocode:**

```
time_to_second_spike = peakttime[1] - stimstart
```

### LibV5 : inv\_time\_to\_first\_spike

1.0 over time to first spike (times 1000 to convert it to Hz); returns 0 when no spike

- **Required features:** time\_to\_first\_spike
- **Units:** Hz
- **Pseudocode:**

```
if len(time_to_first_spike) > 0:  
    inv_time_to_first_spike = 1000.0 / time_to_first_spike[0]  
else:  
    inv_time_to_first_spike = 0
```

### LibV1 : ISI\_values

The interspike intervals (i.e. time intervals) between adjacent peaks. If ignore\_first\_ISI is True, the 1st spike will not be taken into account, because some cells spike right after the stimulus onset and then stay silent for a while.

- **Required features:** peak\_time (ms)
- **Units:** ms
- **Pseudocode:**

```
isi_values = numpy.diff(peak_time)[1:]
```

### LibV1 : doublet\_ISI

The time interval between the first two peaks

- **Required features:** peak\_time (ms)
- **Units:** ms
- **Pseudocode:**

```
doublet_ISI = peak_time[1] - peak_time[0]
```

**LibV5 : all\_ISI\_values**

The interspike intervals, i.e., the time intervals between adjacent peaks.

- **Required features:** peak\_time (ms)
- **Units:** ms
- **Pseudocode:**

```
all_isi_values_vec = numpy.diff(peak_time)
```

**LibV5 : inv\_first\_ISI, inv\_second\_ISI, inv\_third\_ISI, inv\_fourth\_ISI, inv\_fifth\_ISI, inv\_last\_ISI**

1.0 over first/second/third/fourth/fifth/last ISI; returns 0 when no ISI

- **Required features:** peak\_time (ms)
- **Units:** Hz
- **Pseudocode:**

```
all_isi_values_vec = numpy.diff(peak_time)

if len(all_isi_values_vec) > 0:
    inv_first_ISI = 1000.0 / all_isi_values_vec[0]
else:
    inv_first_ISI = 0

if len(all_isi_values_vec) > 1:
    inv_second_ISI = 1000.0 / all_isi_values_vec[1]
else:
    inv_second_ISI = 0

if len(all_isi_values_vec) > 2:
    inv_third_ISI = 1000.0 / all_isi_values_vec[2]
else:
    inv_third_ISI = 0

if len(all_isi_values_vec) > 3:
    inv_fourth_ISI = 1000.0 / all_isi_values_vec[3]
else:
    inv_fourth_ISI = 0

if len(all_isi_values_vec) > 4:
    inv_fifth_ISI = 1000.0 / all_isi_values_vec[4]
else:
    inv_fifth_ISI = 0

if len(all_isi_values_vec) > 0:
    inv_last_ISI = 1000.0 / all_isi_values_vec[-1]
else:
    inv_last_ISI = 0
```

### LibV5 : time\_to\_last\_spike

time from stimulus start to last spike

- **Required features:** peak\_time (ms), stimstart (ms)
- **Units:** ms
- **Pseudocode:**

```
if len(peak_time) > 0:  
    time_to_last_spike = peak_time[-1] - stimstart  
else:  
    time_to_last_spike = 0
```

### LibV1 : Spikecount

number of spikes in the trace, including outside of stimulus interval

- **Required features:** LibV1:peak\_indices
- **Units:** constant
- **Pseudocode:**

```
Spikecount = len(peak_indices)
```

### LibV5 : Spikecount\_stimint

number of spikes inside the stimulus interval

- **Required features:** LibV1:peak\_time
- **Units:** constant
- **Pseudocode:**

```
peaktimes_stimint = numpy.where((peak_time >= stim_start) & (peak_time <= stim_end))  
Spikecount_stimint = len(peaktimes_stimint)
```

### LibV5 : number\_initial\_spikes

number of spikes at the beginning of the stimulus

- **Required features:** LibV1:peak\_time
- **Required parameters:** initial\_perc (default=0.1)
- **Units:** constant
- **Pseudocode:**

```
initial_length = (stimend - stimstart) * initial_perc  
number_initial_spikes = len(numpy.where( \  
    (peak_time >= stimstart) & \  
    (peak_time <= stimstart + initial_length)))
```

### LibV1 : mean\_frequency

The mean frequency of the firing rate

- **Required features:** stim\_start, stim\_end, LibV1:peak\_time
- **Units:** Hz
- **Pseudocode:**

```
condition = np.all((stim_start < peak_time, peak_time < stim_end), axis=0)
spikecount = len(peak_time[condition])
last_spike_time = peak_time[peak_time < stim_end][-1]
mean_frequency = 1000 * spikecount / (last_spike_time - stim_start)
```

### LibV5 : ISI\_semilog\_slope

The slope of a linear fit to a semilog plot of the ISI values.

Attention: the 1st ISI is not taken into account unless ignore\_first\_ISI is set to 0. See LibV1: ISI\_values feature for more details.

- **Required features:** t, V, stim\_start, stim\_end, ISI\_values
- **Units:** ms
- **Pseudocode:**

```
x = range(1, len(ISI_values)+1)
log_ISI_values = numpy.log(ISI_values)
slope, _ = numpy.polyfit(x, log_ISI_values, 1)

ISI_semilog_slope = slope
```

### LibV5 : ISI\_log\_slope

The slope of a linear fit to a loglog plot of the ISI values.

Attention: the 1st ISI is not taken into account unless ignore\_first\_ISI is set to 0. See LibV1: ISI\_values feature for more details.

- **Required features:** t, V, stim\_start, stim\_end, ISI\_values
- **Units:** ms
- **Pseudocode:**

```
log_x = numpy.log(range(1, len(ISI_values)+1))
log_ISI_values = numpy.log(ISI_values)
slope, _ = numpy.polyfit(log_x, log_ISI_values, 1)

ISI_log_slope = slope
```

### LibV5 : ISI\_log\_slope\_skip

The slope of a linear fit to a loglog plot of the ISI values, but not taking into account the first ISI values.

The proportion of ISI values to be skipped is given by `spike_skipf` (between 0 and 1). However, if this number of ISI values to skip is higher than `max_spike_skip`, then `max_spike_skip` is taken instead.

- **Required features:** t, V, stim\_start, stim\_end, ISI\_values
- **Parameters:** spike\_skipf (default=0.1), max\_spike\_skip (default=2)
- **Units:** ms
- **Pseudocode:**

```
start_idx = min([max_spike_skip, round((len(ISI_values) + 1) * spike_skipf)])
ISI_values = ISI_values[start_idx:]
log_x = numpy.log(range(1, len(ISI_values)+1))
log_ISI_values = numpy.log(ISI_values)
slope, _ = numpy.polyfit(log_x, log_ISI_values, 1)

ISI_log_slope = slope
```

### LibV1 : ISI\_CV

The coefficient of variation of the ISIs.

Attention: the 1st ISI is not taken into account unless `ignore_first_ISI` is set to 0. See LibV1: ISI\_values feature for more details.

- **Required features:** ISI\_values
- **Units:** constant
- **Pseudocode:**

```
ISI_mean = numpy.mean(ISI_values)
ISI_variance = numpy.sum(numpy.square(ISI_values-ISI_mean)) / (len(ISI_values)-1)
ISI_std = math.sqrt(ISI_variance)
ISI_CV = ISI_std / ISI_mean
```

### LibV5 : irregularity\_index

Mean of the absolute difference of all ISIs, except the first one (see LibV1: ISI\_values feature for more details.)

The first ISI can be taken into account if `ignore_first_ISI` is set to 0.

- **Required features:** ISI\_values
- **Units:** ms
- **Pseudocode:**

```
irregularity_index = numpy.mean(numpy.absolute(ISI_values[1:] - ISI_values[:-1]))
```



### LibV1 : adaptation\_index

Normalized average difference of two consecutive ISIs, skipping the first ISIs

The proportion of ISI values to be skipped is given by `spike_skipf` (between 0 and 1). However, if this number of ISI values to skip is higher than `max_spike_skip`, then `max_spike_skip` is taken instead.

The adaptation index is zero for a constant firing rate and bigger than zero for a decreasing firing rate

- **Required features:** `stim_start`, `stim_end`, `peak_time`
- **Parameters:** `offset` (default=0), `spike_skipf` (default=0.1), `max_spike_skip` (default=2)
- **Units:** constant
- **Pseudocode:**

```
# skip the first ISIs
peak_selection = [peak_time >= stim_start - offset, peak_time <= stim_end - offset]
spike_time = peak_time[numpy.all(peak_selection, axis=0)]

start_idx = min([max_spike_skip, round(len(spike_time) * spike_skipf)])
spike_time = spike_time[start_idx:]

# compute the adaptation index
ISI_values = spike_time[1:] - spike_time[:-1]
ISI_sum = ISI_values[1:] + ISI_values[:-1]
ISI_sub = ISI_values[1:] - ISI_values[:-1]
adaptation_index = numpy.mean(ISI_sum / ISI_sub)
```

### LibV1 : adaptation\_index\_2

Normalized average difference of two consecutive ISIs, starting at the second ISI

The adaptation index is zero for a constant firing rate and bigger than zero for a decreasing firing rate

- **Required features:** `stim_start`, `stim_end`, `peak_time`
- **Parameters:** `offset` (default=0)
- **Units:** constant
- **Pseudocode:**

```
# skip the first ISI
peak_selection = [peak_time >= stim_start - offset, peak_time <= stim_end - offset]
spike_time = peak_time[numpy.all(peak_selection, axis=0)]

spike_time = spike_time[1:]

# compute the adaptation index
ISI_values = spike_time[1:] - spike_time[:-1]
ISI_sum = ISI_values[1:] + ISI_values[:-1]
ISI_sub = ISI_values[1:] - ISI_values[:-1]
adaptation_index = numpy.mean(ISI_sum / ISI_sub)
```

### LibV5 : check\_AISInitiation

Check initiation of AP in AIS

- **Required features:** t, V, stim\_start, stim\_end, AP\_begin\_time, AP\_begin\_time;location\_AIS
- **Units:** constant
- **Pseudocode:**

```
if len(AP_begin_time) != len(AP_begin_time;location_AIS):
    return None
for soma_time, ais_time in zip(AP_begin_time, AP_begin_time;location_AIS):
    if soma_time < ais_time:
        return None
return [1]
```

### LibV1 : burst\_mean\_freq

The mean frequency during a burst for each burst

If burst\_ISI\_indices did not detect any burst beginning, then the spikes are not considered to be part of any burst

- **Required features:** burst\_ISI\_indices, peak\_time
- **Units:** Hz
- **Pseudocode:**

```
if burst_ISI_indices is None:
    return None
elif len(burst_ISI_indices) == 0:
    return []

burst_mean_freq = []
burst_index = numpy.insert(
    burst_index_tmp, burst_index_tmp.size, len(peak_time) - 1
)

# 1st burst
span = peak_time[burst_index[0]] - peak_time[0]
N_peaks = burst_index[0] + 1
burst_mean_freq.append(N_peaks * 1000 / span)

for i, burst_idx in enumerate(burst_index[:-1]):
    if burst_index[i + 1] - burst_idx != 1:
        span = peak_time[burst_index[i + 1]] - peak_time[burst_idx + 1]
        N_peaks = burst_index[i + 1] - burst_idx
        burst_mean_freq.append(N_peaks * 1000 / span)

return burst_mean_freq
```

**LibV5 : strict\_burst\_mean\_freq**

The mean frequency during a burst for each burst

This implementation does not assume that every spike belongs to a burst.

The first spike is ignored by default. This can be changed by setting `ignore_first_ISI` to 0.

The burst detection can be fine-tuned by changing the setting `strict_burst_factor`. Default value is 2.0.

- **Required features:** `burst_begin_indices`, `burst_end_indices`, `peak_time`
- **Units:** Hz
- **Pseudocode:**

```
if burst_begin_indices is None or burst_end_indices is None:
    strict_burst_mean_freq = None
else:
    strict_burstmean_freq = (
        (burst_end_indices - burst_begin_indices + 1) * 1000 / (
            peak_time[burst_end_indices] - peak_time[burst_begin_indices]
        )
    )
```

**LibV1 : burst\_number**

The number of bursts

- **Required features:** `burst_mean_freq`
- **Units:** constant
- **Pseudocode:**

```
burst_number = len(burst_mean_freq)
```

**LibV5 : strict\_burst\_number**

The number of bursts

This implementation does not assume that every spike belongs to a burst.

The first spike is ignored by default. This can be changed by setting `ignore_first_ISI` to 0.

The burst detection can be fine-tuned by changing the setting `strict_burst_factor`. Default value is 2.0.

- **Required features:** `strict_burst_mean_freq`
- **Units:** constant
- **Pseudocode:**

```
burst_number = len(strict_burst_mean_freq)
```

### LibV1 : interburst\_voltage

The voltage average in between two bursts

Iterating over the burst ISI indices determine the last peak before the burst. Starting 5 ms after that peak take the voltage average until 5 ms before the first peak of the subsequent burst.

- **Required features:** burst\_ISI\_indices, peak\_indices
- **Units:** mV
- **Pseudocode:**

```
interburst_voltage = []
for idx in burst_ISI_idxs:
    ts_idx = peak_idxs[idx]
    t_start = time[ts_idx] + 5
    start_idx = numpy.argwhere(time < t_start)[-1][0]

    te_idx = peak_idxs[idx + 1]
    t_end = time[te_idx] - 5
    end_idx = numpy.argwhere(time > t_end)[0][0]

    interburst_voltage.append(numpy.mean(voltage[start_idx:end_idx + 1]))
```

### LibV5 : strict\_interburst\_voltage

The voltage average in between two bursts

Iterating over the burst indices determine the first peak of each burst. Starting 5 ms after the previous peak, take the voltage average until 5 ms before the peak.

This implementation does not assume that every spike belongs to a burst.

The first spike is ignored by default. This can be changed by setting ignore\_first\_ISI to 0.

The burst detection can be fine-tuned by changing the setting strict\_burst\_factor. Default value is 2.0.

- **Required features:** burst\_begin\_indices, peak\_indices
- **Units:** mV
- **Pseudocode:**

```
interburst_voltage = []
for idx in burst_begin_idxs[1:]:
    ts_idx = peak_idxs[idx - 1]
    t_start = t[ts_idx] + 5
    start_idx = numpy.argwhere(t < t_start)[-1][0]

    te_idx = peak_idxs[idx]
    t_end = t[te_idx] - 5
    end_idx = numpy.argwhere(t > t_end)[0][0]

    interburst_voltage.append(numpy.mean(v[start_idx:end_idx + 1]))
```

### LibV5 : interburst\_min\_values

The minimum voltage between the end of a burst and the next spike.

This implementation does not assume that every spike belongs to a burst.

The first spike is ignored by default. This can be changed by setting `ignore_first_ISI` to 0.

The burst detection can be fine-tuned by changing the setting `strict_burst_factor`. Default value is 2.0.

- **Required features:** `peak_indices`, `burst_end_indices`
- **Units:** mV
- **Pseudocode:**

```
interburst_min = [
    numpy.min(
        v[peak_indices[i]:peak_indices[i + 1]]
    ) for i in burst_end_indices if i + 1 < len(peak_indices)
]
```

### LibV5 : postburst\_min\_values

The minimum voltage after the end of a burst.

This implementation does not assume that every spike belongs to a burst.

The first spike is ignored by default. This can be changed by setting `ignore_first_ISI` to 0.

The burst detection can be fine-tuned by changing the setting `strict_burst_factor`. Default value is 2.0.

- **Required features:** `peak_indices`, `burst_end_indices`
- **Units:** mV
- **Pseudocode:**

```
interburst_min = [
    numpy.min(
        v[peak_indices[i]:peak_indices[i + 1]]
    ) for i in burst_end_indices if i + 1 < len(peak_indices)
]

if len(postburst_min) < len(burst_end_indices):
    if t[burst_end_indices[-1]] < stim_end:
        end_idx = numpy.where(t >= stim_end)[0][0]
        postburst_min.append(numpy.min(
            v[peak_indices[burst_end_indices[-1]]:end_idx]
        ))
    else:
        postburst_min.append(numpy.min(
            v[peak_indices[burst_end_indices[-1]]:]
        ))
```

**LibV5 : time\_to\_interburst\_min**

The time between the last spike of a burst and the minimum between that spike and the next.

This implementation does not assume that every spike belongs to a burst.

The first spike is ignored by default. This can be changed by setting `ignore_first_ISI` to 0.

The burst detection can be fine-tuned by changing the setting `strict_burst_factor`. Default value is 2.0.

- **Required features:** `peak_indices`, `burst_end_indices`, `peak_time`
- **Units:** ms
- **Pseudocode:**

```
time_to_interburst_min = [
    t[peak_indices[i] + numpy.argmin(
        v[peak_indices[i]:peak_indices[i + 1]]
    )] - peak_time[i]
    for i in burst_end_indices if i + 1 < len(peak_indices)
]
```

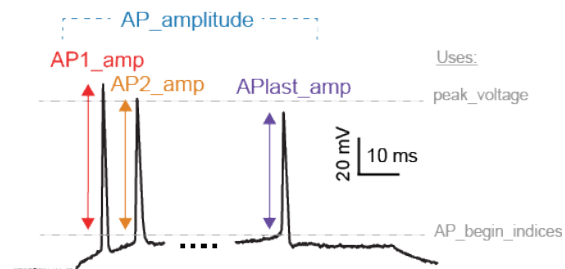
**LibV1 : single\_burst\_ratio**

Length of the second isi over the median of the rest of the isis. The first isi is not taken into account, because it could bias the feature. See LibV1: `ISI_values` feature for more details.

If `ignore_first_ISI` is set to 0, then single burst ratio becomes the length of the first isi over the median of the rest of the isis.

- **Required features:** `ISI_values`
- **Units:** constant
- **Pseudocode:**

```
single_burst_ratio = ISI_values[0] / numpy.mean(ISI_values)
```

**3.1.2 Spike shape features**

**LibV1 : peak\_time**

The times of the maxima of the peaks

- **Required features:** LibV5:peak\_indices
- **Units:** ms
- **Pseudocode:**

```
peak_time = time[peak_indices]
```

**LibV1 : peak\_voltage**

The voltages at the maxima of the peaks

- **Required features:** LibV5:peak\_indices
- **Units:** mV
- **Pseudocode:**

```
peak_voltage = voltage[peak_indices]
```

**LibV1 : AP\_amplitude, AP1\_amp, AP2\_amp, APlast\_amp**

The relative height of the action potential from spike onset

- **Required features:** LibV5:AP\_begin\_indices, LibV1:peak\_voltage (mV)
- **Units:** mV
- **Pseudocode:**

```
AP_amplitude = peak_voltage - voltage[AP_begin_indices]
AP1_amp = AP_amplitude[0]
AP2_amp = AP_amplitude[1]
APlast_amp = AP_amplitude[-1]
```

**LibV5 : mean\_AP\_amplitude**

The mean of all of the action potential amplitudes

- **Required features:** LibV1:AP\_amplitude (mV)
- **Units:** mV
- **Pseudocode:**

```
mean_AP_amplitude = numpy.mean(AP_amplitude)
```

### LibV2 : AP\_Amplitude\_change

Difference of the amplitudes of the second and the first action potential divided by the amplitude of the first action potential

- **Required features:** LibV1:AP\_amplitude
- **Units:** constant
- **Pseudocode:**

```
AP_amplitude_change = (AP_amplitude[1:] - AP_amplitude[0]) / AP_amplitude[0]
```

### LibV5 : AP\_amplitude\_from\_voltagebase

The relative height of the action potential from voltage base

- **Required features:** LibV5:voltage\_base, LibV1:peak\_voltage (mV)
- **Units:** mV
- **Pseudocode:**

```
AP_amplitude_from_voltagebase = peak_voltage - voltage_base
```

### LibV5 : AP1\_peak, AP2\_peak

The peak voltage of the first and second action potentials

- **Required features:** LibV1:peak\_voltage (mV)
- **Units:** mV
- **Pseudocode:**

```
AP1_peak = peak_voltage[0]  
AP2_peak = peak_voltage[1]
```

### LibV5 : AP2\_AP1\_diff

Difference amplitude of the second to first spike

- **Required features:** LibV1:AP\_amplitude (mV)
- **Units:** mV
- **Pseudocode:**

```
AP2_AP1_diff = AP_amplitude[1] - AP_amplitude[0]
```



### LibV5 : AP2\_AP1\_peak\_diff

Difference peak voltage of the second to first spike

- **Required features:** LibV1:peak\_voltage (mV)
- **Units:** mV
- **Pseudocode:**

```
AP2_AP1_diff = peak_voltage[1] - peak_voltage[0]
```

### LibV2 : amp\_drop\_first\_second

Difference of the amplitude of the first and the second peak

- **Required features:** LibV1:peak\_voltage (mV)
- **Units:** mV
- **Pseudocode:**

```
amp_drop_first_second = peak_voltage[0] - peak_voltage[1]
```

### LibV2 : amp\_drop\_first\_last

Difference of the amplitude of the first and the last peak

- **Required features:** LibV1:peak\_voltage (mV)
- **Units:** mV
- **Pseudocode:**

```
amp_drop_first_last = peak_voltage[0] - peak_voltage[-1]
```

### LibV2 : amp\_drop\_second\_last

Difference of the amplitude of the second and the last peak

- **Required features:** LibV1:peak\_voltage (mV)
- **Units:** mV
- **Pseudocode:**

```
amp_drop_second_last = peak_voltage[1] - peak_voltage[-1]
```

**LibV2 : max\_amp\_difference**

Maximum difference of the height of two subsequent peaks

- **Required features:** LibV1:peak\_voltage (mV)
- **Units:** mV
- **Pseudocode:**

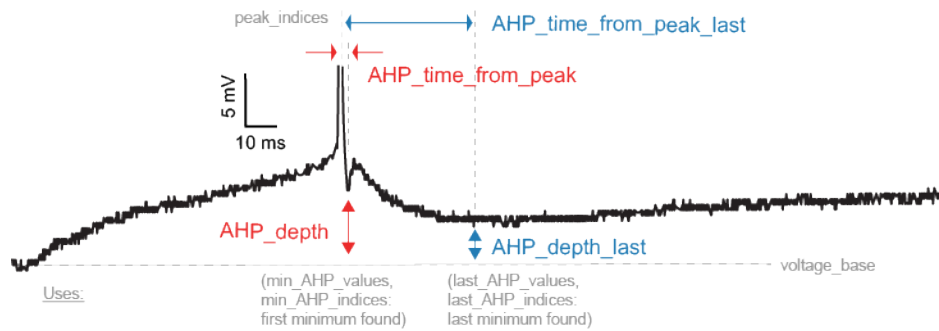
```
max_amp_difference = numpy.max(peak_voltage[:-1] - peak_voltage[1:])
```

**LibV1 : AP\_amplitude\_diff**

Difference of the amplitude of two subsequent peaks

- **Required features:** LibV1:AP\_amplitude (mV)
- **Units:** mV
- **Pseudocode:**

```
AP_amplitude_diff = AP_amplitude[1:] - AP_amplitude[:-1]
```

**LibV5 : min\_AHP\_values**

Absolute voltage values at the first after-hyperpolarization.

- **Required features:** LibV5:min\_AHP\_indices
- **Units:** mV

**LibV5 : AHP\_depth\_abs**

Absolute voltage values at the first after-hyperpolarization. Is the same as min\_AHP\_values

- **Required features:** LibV5:min\_AHP\_values (mV)
- **Units:** mV

**LibV1 : AHP\_depth\_abs\_slow**

Absolute voltage values at the first after-hyperpolarization starting a given number of ms (default: 5) after the peak

- **Required features:** LibV1:peak\_indices
- **Units:** mV

**LibV1 : AHP\_depth\_slow**

Relative voltage values at the first after-hyperpolarization starting a given number of ms (default: 5) after the peak

- **Required features:** LibV5:voltage\_base (mV), LibV1:AHP\_depth\_abs\_slow (mV)
- **Units:** mV
- **Pseudocode:**

```
AHP_depth_slow = AHP_depth_abs_slow[:] - voltage_base
```

**LibV1 : AHP\_slow\_time**

Time difference between slow AHP (see AHP\_depth\_abs\_slow) and peak, divided by interspike interval

- **Required features:** LibV1:AHP\_depth\_abs\_slow
- **Units:** constant

**LibV1 : AHP\_depth**

Relative voltage values at the first after-hyperpolarization

- **Required features:** LibV5:voltage\_base (mV), LibV5:min\_AHP\_values (mV)
- **Units:** mV
- **Pseudocode:**

```
min_AHP_values = first_min_element(voltage, peak_indices)
AHP_depth = min_AHP_values[:] - voltage_base
```

**LibV1 : AHP\_depth\_diff**

Difference of subsequent relative voltage values at the first after-hyperpolarization

- **Required features:** LibV1:AHP\_depth (mV)
- **Units:** mV
- **Pseudocode:**

```
AHP_depth_diff = AHP_depth[1:] - AHP_depth[:-1]
```

### LibV2 : fast\_AHP

Voltage value of the action potential onset relative to the subsequent AHP

Ignores the last spike

- **Required features:** LibV5:AP\_begin\_indices, LibV5:min\_AHP\_values
- **Units:** mV
- **Pseudocode:**

```
fast_AHP = voltage[AP_begin_indices[:-1]] - voltage[min_AHP_indices[:-1]]
```

### LibV2 : fast\_AHP\_change

Difference of the fast AHP of the second and the first action potential divided by the fast AHP of the first action potential

- **Required features:** LibV2:fast\_AHP
- **Units:** constant
- **Pseudocode:**

```
fast_AHP_change = (fast_AHP[1:] - fast_AHP[0]) / fast_AHP[0]
```

### LibV5 : AHP\_depth\_from\_peak, AHP1\_depth\_from\_peak, AHP2\_depth\_from\_peak

Voltage difference between AP peaks and first AHP depths

- **Required features:** LibV1:peak\_indices, LibV5:min\_AHP\_indices
- **Units:** mV
- **Pseudocode:**

```
AHP_depth_from_peak = v[peak_indices] - v[min_AHP_indices]
AHP1_depth_from_peak = AHP_depth_from_peak[0]
AHP2_depth_from_peak = AHP_depth_from_peak[1]
```

### LibV5 : AHP\_time\_from\_peak

Time between AP peaks and first AHP depths

- **Required features:** LibV1:peak\_indices, LibV5:min\_AHP\_values (mV)
- **Units:** ms
- **Pseudocode:**

```
min_AHP_indices = first_min_element(voltage, peak_indices)
AHP_time_from_peak = t[min_AHP_indices[:]] - t[peak_indices[i]]
```

### LibV5 : ADP\_peak\_values

Absolute voltage values of the small afterdepolarization peak

strict\_stiminterval should be set to True for this feature to behave as expected.

- **Required features:** LibV5:min\_AHP\_indices, LibV5:min\_between\_peaks\_indices
- **Units:** mV
- **Pseudocode:**

```
adp_peak_values = numpy.array(
    [numpy.max(v[i:j + 1]) for (i, j) in zip(min_AHP_indices, min_v_indices)]
)
```

### LibV5 : ADP\_peak\_amplitude

Amplitude of the small afterdepolarization peak with respect to the fast AHP voltage

strict\_stiminterval should be set to True for this feature to behave as expected.

- **Required features:** LibV5:min\_AHP\_values, LibV5:ADP\_peak\_values
- **Units:** mV
- **Pseudocode:**

```
adp_peak_amplitude = adp_peak_values - min_AHP_values
```

### LibV3 : depolarized\_base

Mean voltage between consecutive spikes (from the end of one spike to the beginning of the next one)

- **Required features:** LibV5:AP\_end\_indices, LibV5:AP\_begin\_indices
- **Units:** mV
- **Pseudocode:**

```
depolarized_base = []
for (start_idx, end_idx) in zip(
    AP_end_indices[:-1], AP_begin_indices[1:])
):
    depolarized_base.append(numpy.mean(voltage[start_idx:end_idx]))
```

### LibV5 : min\_voltage\_between\_spikes

Minimal voltage between consecutive spikes

- **Required features:** LibV5:peak\_indices
- **Units:** mV
- **Pseudocode:**

```

min_voltage_between_spikes = []
for peak1, peak2 in zip(peak_indices[:-1], peak_indices[1:]):
    min_voltage_between_spikes.append(numpy.min(voltage[peak1:peak2]))

```

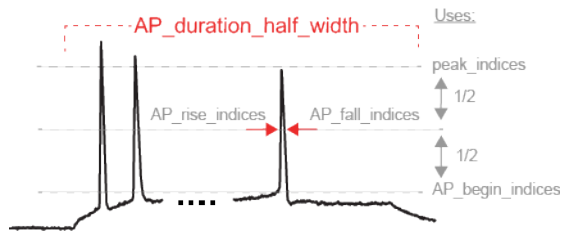
### LibV5 : min\_between\_peaks\_values

Minimal voltage between consecutive spikes

The last value of min\_between\_peaks\_values is the minimum between last spike and stimulus end if strict stiminterval is True, and minimum between last spike and last voltage value if strict stiminterval is False

- **Required features:** LibV5:min\_between\_peaks\_indices
- **Units:** mV
- **Pseudocode:**

```
min_between_peaks_values = v[min_between_peaks_indices]
```



### LibV2 : AP\_duration\_half\_width

Width of spike at half spike amplitude, with spike onset as described in LibV5: AP\_begin\_time

- **Required features:** LibV2: AP\_rise\_indices, LibV2: AP\_fall\_indices
- **Units:** ms
- **Pseudocode:**

```

AP_rise_indices = index_before_peak((v(peak_indices) - v(AP_begin_indices)) / 2)
AP_fall_indices = index_after_peak((v(peak_indices) - v(AP_begin_indices)) / 2)
AP_duration_half_width = t(AP_fall_indices) - t(AP_rise_indices)

```

### LibV2 : AP\_duration\_half\_width\_change

Difference of the FWHM of the second and the first action potential divided by the FWHM of the first action potential

- **Required features:** LibV2: AP\_duration\_half\_width
- **Units:** constant
- **Pseudocode:**

```

AP_duration_half_width_change = (
    AP_duration_half_width[1:] - AP_duration_half_width[0]
) / AP_duration_half_width[0]

```

### LibV1 : AP\_width

Width of spike at threshold, bounded by minimum AHP

Can use `strict_stiminterval` compute only for data in stimulus interval.

- **Required features:** LibV1: `peak_indices`, LibV5: `min_AHP_indices`, `threshold`
- **Units:** ms
- **Pseudocode:**

```

min_AHP_indices = numpy.concatenate([[stim_start], min_AHP_indices])
for i in range(len(min_AHP_indices)-1):
    onset_index = numpy.where(v[min_AHP_indices[i]:min_AHP_indices[i+1]] >_
↳threshold)[0]
    onset_time[i] = t[onset_index]
    offset_index = numpy.where(v[onset_index:min_AHP_indices[i+1]] <_
↳threshold)[0]
    AP_width[i] = t[offset_index] - t[onset_index]

```

### LibV2 : AP\_duration

Duration of an action potential from onset to offset

- **Required features:** LibV5: `AP_begin_indices`, LibV5: `AP_end_indices`
- **Units:** ms
- **Pseudocode:**

```

AP_duration = time[AP_end_indices] - time[AP_begin_indices]

```

### LibV2 : AP\_duration\_change

Difference of the durations of the second and the first action potential divided by the duration of the first action potential

- **Required features:** LibV2: `AP_duration`
- **Units:** constant
- **Pseudocode:**

```

AP_duration_change = (AP_duration[1:] - AP_duration[0]) / AP_duration[0]

```

### LibV5 : AP\_width\_between\_threshold

Width of spike at threshold, bounded by minimum between peaks

Can use `strict_stiminterval` to not use minimum after stimulus end.

- **Required features:** LibV1: `peak_indices`, LibV5: `min_between_peaks_indices`, `threshold`
- **Units:** ms
- **Pseudocode:**

```

min_between_peaks_indices = numpy.concatenate([[stim_start], min_between_peaks_
↳indices])
for i in range(len(min_between_peaks_indices)-1):
    onset_index = numpy.where(v[min_between_peaks_indices[i]:min_between_peaks_
↳indices[i+1]] > threshold)[0]
    onset_time[i] = t[onset_index]
    offset_time[i] = t[numpy.where(v[onset_index:min_between_peaks_indices[i+1]] <↳
↳threshold)[0]]
    AP_width[i] = t(offset_time[i]) - t(onset_time[i])

```

### LibV5 : spike\_half\_width, AP1\_width, AP2\_width, APlast\_width

Width of spike at half spike amplitude, with the spike amplitude taken as the difference between the minimum between two peaks and the next peak

- **Required features:** LibV5: peak\_indices, LibV5: min\_AHP\_indices
- **Units:** ms
- **Pseudocode:**

```

min_AHP_indices = numpy.concatenate([[stim_start], min_AHP_indices])
for i in range(1, len(min_AHP_indices)):
    v_half_width = (v[peak_indices[i-1]] + v[min_AHP_indices[i]]) / 2.
    rise_idx = numpy.where(v[min_AHP_indices[i-1]:peak_indices[i-1]] > v_half_
↳width)[0]
    v_dev = v_half_width - v[rise_idx]
    delta_v = v[rise_idx] - v[rise_idx - 1]
    delta_t = t[rise_idx] - t[rise_idx - 1]
    t_dev_rise = delta_t * v_dev / delta_v

    fall_idx = numpy.where(v[peak_indices[i-1]:min_AHP_indices[i]] < v_half_
↳width)[0]
    v_dev = v_half_width - v[fall_idx]
    delta_v = v[fall_idx] - v[fall_idx - 1]
    delta_t = t[fall_idx] - t[fall_idx - 1]
    t_dev_fall = delta_t * v_dev / delta_v
    spike_half_width[i] = t[fall_idx] + t_dev_fall - t[rise_idx] - t_dev_rise

AP1_width = spike_half_width[0]
AP2_width = spike_half_width[1]
APlast_width = spike_half_width[-1]

```

### LibV1 : spike\_width2

Width of spike at half spike amplitude, with the spike onset taken as the maximum of the second derivative of the voltage in the range between the minimum between two peaks and the next peak

- **Required features:** LibV5: peak\_indices, LibV5: min\_AHP\_indices
- **Units:** ms
- **Pseudocode:**



```

for i in range(len(min_AHP_indices)):
    dv2 = CentralDiffDerivative(CentralDiffDerivative(v[min_AHP_indices[i]:peak_
↪indices[i + 1]]))
    peak_onset_idx = numpy.argmax(dv2) + min_AHP_indices[i]
    v_half_width = (v[peak_indices[i + 1]] + v[peak_onset_idx]) / 2.

    rise_idx = numpy.where(v[peak_onset_idx:peak_indices[i + 1]] > v_half_width)[0]
    v_dev = v_half_width - v[rise_idx]
    delta_v = v[rise_idx] - v[rise_idx - 1]
    delta_t = t[rise_idx] - t[rise_idx - 1]
    t_dev_rise = delta_t * v_dev / delta_v

    fall_idx = numpy.where(v[peak_indices[i + 1]:] < v_half_width)[0]
    v_dev = v_half_width - v[fall_idx]
    delta_v = v[fall_idx] - v[fall_idx - 1]
    delta_t = t[fall_idx] - t[fall_idx - 1]
    t_dev_fall = delta_t * v_dev / delta_v
    spike_width2[i] = t[fall_idx] + t_dev_fall - t[rise_idx] - t_dev_rise

```

### LibV5 : AP\_begin\_width, AP1\_begin\_width, AP2\_begin\_width

Width of spike at spike start

- **Required features:** LibV5: min\_AHP\_indices, LibV5: AP\_begin\_indices
- **Units:** ms
- **Pseudocode:**

```

for i in range(len(min_AHP_indices)):
    rise_idx = AP_begin_indices[i]
    fall_idx = numpy.where(v[rise_idx + 1:min_AHP_indices[i]] < v[rise_idx])[0]
    AP_begin_width[i] = t[fall_idx] - t[rise_idx]

AP1_begin_width = AP_begin_width[0]
AP2_begin_width = AP_begin_width[1]

```

### LibV5 : AP2\_AP1\_begin\_width\_diff

Difference width of the second to first spike

- **Required features:** LibV5: AP\_begin\_width
- **Units:** ms
- **Pseudocode:**

```

AP2_AP1_begin_width_diff = AP_begin_width[1] - AP_begin_width[0]

```

### LibV5 : AP\_begin\_voltage, AP1\_begin\_voltage, AP2\_begin\_voltage

Voltage at spike start

- **Required features:** LibV5: AP\_begin\_indices
- **Units:** mV
- **Pseudocode:**

```
AP_begin_voltage = v[AP_begin_indices]
AP1_begin_voltage = AP_begin_voltage[0]
AP2_begin_voltage = AP_begin_voltage[1]
```

### LibV5 : AP\_begin\_time

Time at spike start. Spike start is defined as where the first derivative of the voltage trace is higher than 10 V/s , for at least 5 points

- **Required features:** LibV5: AP\_begin\_indices
- **Units:** ms
- **Pseudocode:**

```
AP_begin_time = t[AP_begin_indices]
```

### LibV5 : AP\_peak\_upstroke

Maximum of rise rate of spike

- **Required features:** LibV5: AP\_begin\_indices, LibV5: peak\_indices
- **Units:** V/s
- **Pseudocode:**

```
ap_peak_upstroke = []
for apbi, pi in zip(ap_begin_indices, peak_indices):
    ap_peak_upstroke.append(numpy.max(dvdt[apbi:pi]))
```

### LibV5 : AP\_peak\_downstroke

Minimum of fall rate from spike

- **Required features:** LibV5: min\_AHP\_indices, LibV5: peak\_indices
- **Units:** V/s
- **Pseudocode:**

```
ap_peak_downstroke = []
for ahpi, pi in zip(min_ahp_indices, peak_indices):
    ap_peak_downstroke.append(numpy.min(dvdt[pi:ahpi]))
```

### LibV2 : AP\_rise\_time

Time between the AP threshold and the peak, given a window (default: from 0% to 100% of the AP amplitude)

- **Required features:** LibV5: AP\_begin\_indices, LibV5: peak\_indices, LibV1: AP\_amplitude
- **Units:** ms
- **Pseudocode:**

```

rise_times = []
begin_voltages = AP_amps * rise_start_perc + voltage[AP_begin_indices]
end_voltages = AP_amps * rise_end_perc + voltage[AP_begin_indices]

for AP_begin_indice, peak_indice, begin_v, end_v in zip(
    AP_begin_indices, peak_indices, begin_voltages, end_voltages
):
    voltage_window = voltage[AP_begin_indice:peak_indice]

    new_begin_indice = AP_begin_indice + numpy.min(
        numpy.where(voltage_window >= begin_v)[0]
    )
    new_end_indice = AP_begin_indice + numpy.max(
        numpy.where(voltage_window <= end_v)[0]
    )

    rise_times.append(time[new_end_indice] - time[new_begin_indice])

```

### LibV2 : AP\_fall\_time

Time from action potential maximum to the offset

- **Required features:** LibV5: AP\_end\_indices, LibV5: peak\_indices
- **Units:** ms
- **Pseudocode:**

```

AP_fall_time = time[AP_end_indices] - time[peak_indices]

```

### LibV2 : AP\_rise\_rate

Voltage change rate during the rising phase of the action potential

- **Required features:** LibV5: AP\_begin\_indices, LibV5: peak\_indices
- **Units:** V/s
- **Pseudocode:**

```

AP_rise_rate = (voltage[peak_indices] - voltage[AP_begin_indices]) / (
    time[peak_indices] - time[AP_begin_indices]
)

```

### LibV2 : AP\_fall\_rate

Voltage change rate during the falling phase of the action potential

- **Required features:** LibV5: AP\_end\_indices, LibV5: peak\_indices
- **Units:** V/s
- **Pseudocode:**

```
AP_fall_rate = (voltage[AP_end_indices] - voltage[peak_indices]) / (
    time[AP_end_indices] - time[peak_indices]
)
```

### LibV2 : AP\_rise\_rate\_change

Difference of the rise rates of the second and the first action potential divided by the rise rate of the first action potential

- **Required features:** LibV2: AP\_rise\_rate\_change
- **Units:** constant
- **Pseudocode:**

```
AP_rise_rate_change = (AP_rise_rate[1:] - AP_rise_rate[0]) / AP_rise_rate[0]
```

### LibV2 : AP\_fall\_rate\_change

Difference of the fall rates of the second and the first action potential divided by the fall rate of the first action potential

- **Required features:** LibV2: AP\_fall\_rate\_change
- **Units:** constant
- **Pseudocode:**

```
AP_fall_rate_change = (AP_fall_rate[1:] - AP_fall_rate[0]) / AP_fall_rate[0]
```

### LibV5 : AP\_phaseslope

Slope of the V, dV/dt phasespace plot at the beginning of every spike

(at the point where the derivative crosses the DerivativeThreshold)

- **Required features:** LibV5:AP\_begin\_indices
- **Parameters:** AP\_phaseslope\_range
- **Units:** 1/(ms)
- **Pseudocode:**

```
range_max_idx = AP_begin_indices + AP_phseslope_range
range_min_idx = AP_begin_indices - AP_phseslope_range
AP_phaseslope = (dvdt[range_max_idx] - dvdt[range_min_idx]) / (v[range_max_idx] -
    ↪ v[range_min_idx])
```

### LibV5 : AP\_phaseslope\_AIS

Same as AP\_phaseslope, but for AIS location

Please, notice that you have to provide t, v, stim\_start and stim\_end for location.

- **Required features:** T;location\_AIS, V;location\_AIS, stim\_start;location\_AIS, stim\_end;location\_AIS, LibV5:AP\_begin\_indices;location\_AIS
- **Parameters:** AP\_phaseslope\_range
- **Units:** 1/(ms)
- **Pseudocode:**

```
range_max_idx = AP_begin_indices + AP_phseslope_range
range_min_idx = AP_begin_indices - AP_phseslope_range
AP_phaseslope_AIS = (dvdtd[range_max_idx] - dvdtd[range_min_idx]) / (v[range_max_
↪idx] - v[range_min_idx])
```

### LibV5 : BPAPHeightLoc1

Voltage height (difference between peaks and voltage base) at dendrite location

Please, notice that you have to provide t, v, stim\_start and stim\_end for location.

- **Required features:** T;location\_dend1, V;location\_dend1, stim\_start;location\_dend1, stim\_end;location\_dend1, peak\_voltage;location\_dend1, voltage\_base;location\_dend1
- **Units:** mV
- **Pseudocode:**

```
BPAPHeightLoc1 = peak_voltage - voltage_base
```

### LibV5 : BPAPHeightLoc2

Same as BPAPHeightLoc1, but for dend2 location

- **Required features:** T;location\_dend2, V;location\_dend2, stim\_start;location\_dend2, stim\_end;location\_dend2, peak\_voltage;location\_dend2, voltage\_base;location\_dend2
- **Units:** mV
- **Pseudocode:**

```
BPAPHeightLoc2 = peak_voltage - voltage_base
```

### LibV5 : BPAPAmplitudeLoc1

Amplitude at dendrite location

Please, notice that you have to provide t, v, stim\_start and stim\_end for location.

- **Required features:** T;location\_dend1, V;location\_dend1, stim\_start;location\_dend1, stim\_end;location\_dend1, peak\_voltage;location\_dend1, AP\_begin\_voltage;location\_dend1
- **Units:** mV
- **Pseudocode:**

```
BPAPAmplitudeLoc1 = peak_voltage - AP_begin_voltage
```

### LibV5 : BPAPAmplitudeLoc2

Same as BPAPAmplitudeLoc1, but for dend2 location

- **Required features:** T;location\_dend2, V;location\_dend2, stim\_start;location\_dend2, stim\_end;location\_dend2, peak\_voltage;location\_dend2, AP\_begin\_voltage;location\_dend2
- **Units:** mV
- **Pseudocode:**

```
BPAPAmplitudeLoc2 = peak_voltage - AP_begin_voltage
```

### LibV5 : BAC\_width

AP width at epsp location

Please, notice that you have to provide t, v, stim\_start and stim\_end for location.

- **Required features:** T;location\_epsp, V;location\_epsp, stim\_start;location\_epsp, stim\_end;location\_epsp, AP\_width;location\_epsp
- **Units:** ms
- **Pseudocode:**

```
BAC_width = AP_width
```

### LibV5 : BAC\_maximum\_voltage

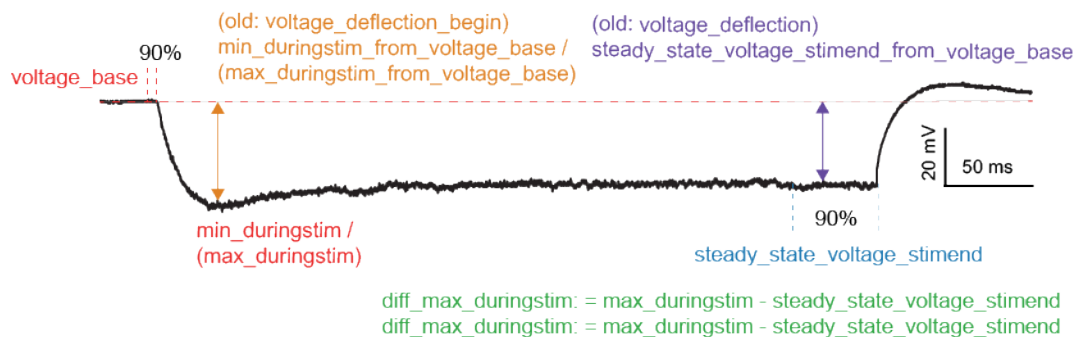
Maximum voltage at epsp location

Please, notice that you have to provide t, v, stim\_start and stim\_end for location.

- **Required features:** T;location\_epsp, V;location\_epsp, stim\_start;location\_epsp, stim\_end;location\_epsp, maximum\_voltage;location\_epsp
- **Units:** mV
- **Pseudocode:**

```
BAC_maximum_voltage = maximum_voltage
```

### 3.1.3 Voltage features



#### LibV5 : steady\_state\_voltage\_stimend

The average voltage during the last 10% of the stimulus duration.

- **Required features:** t, V, stim\_start, stim\_end
- **Units:** mV
- **Pseudocode:**

```
stim_duration = stim_end - stim_start
begin_time = stim_end - 0.1 * stim_duration
end_time = stim_end
steady_state_voltage_stimend = numpy.mean(voltage[numpy.where((t < end_time) & (t >
↪= begin_time))])
```

#### LibV2 : steady\_state\_hyper

Steady state voltage during hyperpolarization for 30 data points (after interpolation)

- **Required features:** t, V, stim\_start, stim\_end
- **Units:** mV
- **Pseudocode:**

```
stim_end_idx = numpy.argmax(time >= stim_end)[0][0]
steady_state_hyper = numpy.mean(voltage[stim_end_idx - 35:stim_end_idx - 5])
```

#### LibV1 : steady\_state\_voltage

The average voltage after the stimulus

- **Required features:** t, V, stim\_end
- **Units:** mV
- **Pseudocode:**

```
steady_state_voltage = numpy.mean(voltage[numpy.where((t <= max(t)) & (t > stim_
↪end))])
```

### LibV5 : voltage\_base

The average voltage during the last 10% of time before the stimulus.

- **Required features:** t, V, stim\_start, stim\_end
- **Parameters:** voltage\_base\_start\_perc (default = 0.9), voltage\_base\_end\_perc (default = 1.0)
- **Units:** mV
- **Pseudocode:**

```
voltage_base = numpy.mean(voltage[numpy.where(
    (t >= voltage_base_start_perc * stim_start) &
    (t <= voltage_base_end_perc * stim_start))])
```

### LibV5 : current\_base

The average current during the last 10% of time before the stimulus.

- **Required features:** t, I, stim\_start, stim\_end
- **Parameters:** current\_base\_start\_perc (default = 0.9), current\_base\_end\_perc (default = 1.0), precision\_threshold (default = 1e-10), current\_base\_mode (can be “mean” or “median”, default=“mean”)
- **Units:** nA
- **Pseudocode:**

```
current_slice = I[numpy.where(
    (t >= current_base_start_perc * stim_start) &
    (t <= current_base_end_perc * stim_start))]
if current_base_mode == "mean":
    current_base = numpy.mean(current_slice)
elif current_base_mode == "median":
    current_base = numpy.median(current_slice)
```

### LibV1 : time\_constant

The membrane time constant

The extraction of the time constant requires a voltage trace of a cell in a hyper-polarized state. Starting at stim start find the beginning of the exponential decay where the first derivative of V(t) is smaller than -0.005 V/s in 5 subsequent points. The flat subsequent to the exponential decay is defined as the point where the first derivative of the voltage trace is bigger than -0.005 and the mean of the following 70 points as well. If the voltage trace between the beginning of the decay and the flat includes more than 9 points, fit an exponential decay. Yield the time constant of that decay.

- **Required features:** t, V, stim\_start, stim\_end
- **Units:** ms
- **Pseudocode:**

```
min_derivative = 5e-3
decay_start_min_length = 5 # number of indices
min_length = 10 # number of indices
t_length = 70 # in ms
```

(continues on next page)



(continued from previous page)

```

# get start and middle indices
stim_start_idx = numpy.where(time >= stim_start)[0][0]
# increment stimstartindex to skip a possible transient
stim_start_idx += 10
stim_middle_idx = numpy.where(time >= (stim_start + stim_end) / 2.0)[0][0]

# get derivative
t_interval = time[stim_start_idx:stim_middle_idx]
dv = five_point_stencil_derivative(voltage[stim_start_idx:stim_middle_idx])
dt = five_point_stencil_derivative(t_interval)
dvdt = dv / dt

# find start and end of decay
# has to be over deriv threshold for at least a given number of indices
pass_threshold_idx = numpy.append(
    -1, numpy.argwhere(dvdt > -min_derivative).flatten()
)
length_idx = numpy.argwhere(
    numpy.diff(pass_threshold_idx) > decay_start_min_length
)[0][0]
i_start = pass_threshold_idx[length_idx] + 1

# find flat (end of decay)
flat_idx = numpy.argwhere(dvdt[i_start:] > -min_derivative).flatten()
# for loop is not optimised
# but we expect the 1st few values to be the ones we are looking for
for i in flat_idx:
    i_flat = i + i_start
    i_flat_stop = numpy.argwhere(
        t_interval >= t_interval[i_flat] + t_length
    )[0][0]
    if numpy.mean(dvdt[i_flat:i_flat_stop]) > -min_derivative:
        break

dvdt_decay = dvdt[i_start:i_flat]
t_decay = time[stim_start_idx + i_start:stim_start_idx + i_flat]
v_decay_tmp = voltage[stim_start_idx + i_start:stim_start_idx + i_flat]
v_decay = abs(v_decay_tmp - voltage[stim_start_idx + i_flat])

if len(dvdt_decay) < min_length:
    return None

# -- golden search algorithm -- #
from scipy.optimize import minimize_scalar

def numpy_fit(x, t_decay, v_decay):
    new_v_decay = v_decay + x
    log_v_decay = numpy.log(new_v_decay)
    (slope, _), res, _, _, _ = numpy.polyfit(
        t_decay, log_v_decay, 1, full=True
    )

```

(continues on next page)

(continued from previous page)

```

range = numpy.max(log_v_decay) - numpy.min(log_v_decay)
return res / (range * range)

max_bound = min_derivative * 1000.
golden_bracket = [0, max_bound]
result = minimize_scalar(
    numpy_fit,
    args=(t_decay, v_decay),
    bracket=golden_bracket,
    method='golden',
)

# -- fit -- #
log_v_decay = numpy.log(v_decay + result.x)
slope, _ = numpy.polyfit(t_decay, log_v_decay, 1)

time_constant = -1. / slope

```

### LibV5 : decay\_time\_constant\_after\_stim

The decay time constant of the voltage right after the stimulus

- **Required features:** t, V, stim\_start, stim\_end
- **Parameters:** decay\_start\_after\_stim (default = 1.0 ms), decay\_end\_after\_stim (default = 10.0 ms)
- **Units:** ms
- **Pseudocode:**

```

time_interval = t[numpy.where(t => decay_start_after_stim &
    t < decay_end_after_stim)] - t[numpy.where(t == stim_end)]
voltage_interval = abs(voltages[numpy.where(t => decay_start_after_stim &
    t < decay_end_after_stim)]
    - voltages[numpy.where(t == decay_start_after_stim)])

log_voltage_interval = numpy.log(voltage_interval)
slope, _ = numpy.polyfit(time_interval, log_voltage_interval, 1)

decay_time_constant_after_stim = -1. / slope

```

### LibV5 : multiple\_decay\_time\_constant\_after\_stim

When multiple stimuli are applied, this function returns a list of decay time constants each computed on the voltage right after each stimulus.

The settings multi\_stim\_start and multi\_stim\_end are mandatory for this feature to work. Each is a list containing the start and end times of each stimulus present in the current protocol respectively.

- **Required features:** t, V, stim\_start, stim\_end
- **Required settings:** multi\_stim\_start, multi\_stim\_end
- **Parameters:** decay\_start\_after\_stim (default = 1.0 ms), decay\_end\_after\_stim (default = 10.0 ms)

- **Units:** ms
- **Pseudocode:**

```
multiple_decay_time_constant_after_stim = []
for i in range(len(number_stimuli):
    stim_start = multi_stim_start[i]
    stim_end = multi_stim_end[i]
    multiple_decay_time_constant_after_stim.append(
        decay_time_constant_after_stim(stim_start, stim_end)
    )
```

### LibV5 : sag\_time\_constant

The decay time constant of the exponential voltage decay from the bottom of the sag to the steady-state.

The start of the decay is taken at the minimum voltage (the bottom of the sag). The end of the decay is taken when the voltage crosses the steady state voltage minus 10% of the sag amplitude. The time constant is the slope of the linear fit to the log of the voltage. The golden search algorithm is not used, since the data is expected to be noisy and adding a parameter in the log (  $\log(\text{voltage} + x)$  ) is likely to increase errors on the fit.

- **Required features:** t, V, stim\_start, stim\_end, minimum\_voltage, steady\_state\_voltage\_stimend, sag\_amplitude
- **Units:** ms
- **Pseudocode:**

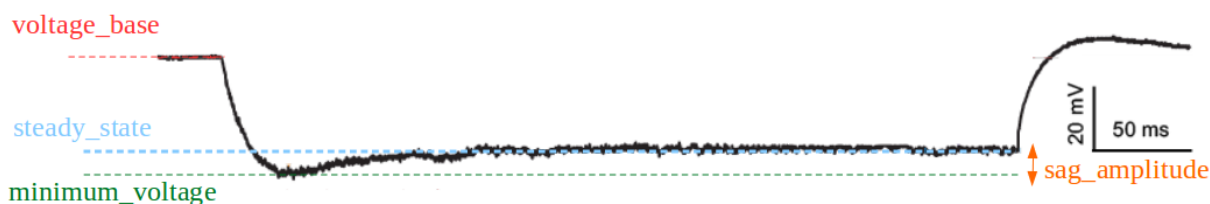
```
# get start decay
start_decay = numpy.argmin(vinterval)

# get end decay
v90 = steady_state_v - 0.1 * sag_ampl
end_decay = numpy.where((tinterval > tinterval[start_decay]) & (vinterval >=
→v90))[0][0]

v_reference = vinterval[end_decay]

# select t, v in decay interval
interval_indices = numpy.arange(start_decay, end_decay)
interval_time = tinterval[interval_indices]
interval_voltage = abs(vinterval[interval_indices] - v_reference)

# get tau
log_interval_voltage = numpy.log(interval_voltage)
slope, _ = numpy.polyfit(interval_time, log_interval_voltage, 1)
tau = abs(1. / slope)
```



### LibV5 : sag\_amplitude

The difference between the minimal voltage and the steady state at stimend

- **Required features:** t, V, stim\_start, stim\_end, steady\_state\_voltage\_stimend, minimum\_voltage, voltage\_deflection\_stim\_ssse
- **Parameters:**
- **Units:** mV
- **Pseudocode:**

```
if (voltage_deflection_stim_ssse <= 0):
    sag_amplitude = steady_state_voltage_stimend - minimum_voltage
else:
    sag_amplitude = None
```

### LibV5 : sag\_ratio1

The ratio between the sag amplitude and the maximal sag extend from voltage base

- **Required features:** t, V, stim\_start, stim\_end, sag\_amplitude, voltage\_base, minimum\_voltage
- **Parameters:**
- **Units:** constant
- **Pseudocode:**

```
if voltage_base != minimum_voltage:
    sag_ratio1 = sag_amplitude / (voltage_base - minimum_voltage)
else:
    sag_ratio1 = None
```

### LibV5 : sag\_ratio2

The ratio between the maximal extends of sag from steady state and voltage base

- **Required features:** t, V, stim\_start, stim\_end, steady\_state\_voltage\_stimend, voltage\_base, minimum\_voltage
- **Parameters:**
- **Units:** constant
- **Pseudocode:**

```
if voltage_base != minimum_voltage:
    sag_ratio2 = (voltage_base - steady_state_voltage_stimend) / (voltage_base -
    ↪minimum_voltage)
else:
    sag_ratio2 = None
```

### LibV1 : ohmic\_input\_resistance

The ratio between the voltage deflection and stimulus current

- **Required features:** t, V, stim\_start, stim\_end, voltage\_deflection
- **Parameters:** stimulus\_current
- **Units:** M
- **Pseudocode:**

```
ohmic_input_resistance = voltage_deflection / stimulus_current
```

### LibV5 : ohmic\_input\_resistance\_vb\_ssse

The ratio between the voltage deflection (between voltage base and steady-state voltage at stimend) and stimulus current

- **Required features:** t, V, stim\_start, stim\_end, voltage\_deflection\_vb\_ssse
- **Parameters:** stimulus\_current
- **Units:** M
- **Pseudocode:**

```
ohmic_input_resistance_vb_ssse = voltage_deflection_vb_ssse / stimulus_current
```

### LibV5 : voltage\_deflection\_vb\_ssse

The voltage deflection between voltage base and steady-state voltage at stimend

The voltage base used is the average voltage during the last 10% of time before the stimulus and the steady state voltage at stimend used is the average voltage during the last 10% of the stimulus duration.

- **Required features:** t, V, stim\_start, stim\_end, voltage\_base, steady\_state\_voltage\_stimend
- **Units:** mV
- **Pseudocode:**

```
voltage_deflection_vb_ssse = steady_state_voltage_stimend - voltage_base
```

### LibV1 : voltage\_deflection

The voltage deflection between voltage base and steady-state voltage at stimend

The voltage base used is the average voltage during all of the time before the stimulus and the steady state voltage at stimend used is the average voltage of the five values before the last five values before the end of the stimulus duration.

- **Required features:** t, V, stim\_start, stim\_end
- **Units:** mV
- **Pseudocode:**

```
voltage_base = numpy.mean(V[t < stim_start])
stim_end_idx = numpy.where(t >= stim_end)[0][0]
steady_state_voltage_stimend = numpy.mean(V[stim_end_idx-10:stim_end_idx-5])
voltage_deflection = steady_state_voltage_stimend - voltage_base
```

### LibV5 : voltage\_deflection\_begin

The voltage deflection between voltage base and steady-state voltage soon after stimulation start

The voltage base used is the average voltage during all of the time before the stimulus and the steady state voltage used is the average voltage taken from 5% to 15% of the stimulus duration.

- **Required features:** t, V, stim\_start, stim\_end
- **Units:** mV
- **Pseudocode:**

```
voltage_base = numpy.mean(V[t < stim_start])
tstart = stim_start + 0.05 * (stim_end - stim_start)
tend = stim_start + 0.15 * (stim_end - stim_start)
condition = numpy.all((tstart < t, t < tend), axis=0)
steady_state_voltage_stimend = numpy.mean(V[condition])
voltage_deflection = steady_state_voltage_stimend - voltage_base
```

### LibV5 : voltage\_after\_stim

The mean voltage after the stimulus in (stim\_end + 25%\*end\_period, stim\_end + 75%\*end\_period)

- **Required features:** t, V, stim\_end
- **Units:** mV
- **Pseudocode:**

```
tstart = stim_end + (t[-1] - stimEnd) * 0.25
tend = stim_end + (t[-1] - stimEnd) * 0.75
condition = numpy.all((tstart < t, t < tend), axis=0)
voltage_after_stim = numpy.mean(V[condition])
```

### LibV1 : minimum\_voltage

The minimum of the voltage during the stimulus

- **Required features:** t, V, stim\_start, stim\_end
- **Units:** mV
- **Pseudocode:**

```
minimum_voltage = min(voltage[numpy.where((t >= stim_start) & (t <= stim_end))])
```

### LibV1 : maximum\_voltage

The maximum of the voltage during the stimulus

- **Required features:** t, V, stim\_start, stim\_end
- **Units:** mV
- **Pseudocode:**

```
maximum_voltage = max(voltage[numpy.where((t >= stim_start) & (t <= stim_end))])
```

### LibV5 : maximum\_voltage\_from\_voltagebase

Difference between maximum voltage during stimulus and voltage base

- **Required features:** maximum\_voltage, voltage\_base
- **Units:** mV
- **Pseudocode:**

```
maximum_voltage_from_voltagebase = maximum_voltage - voltage_base
```

## 3.2 Requested eFeatures

### 3.2.1 Cpp features

#### LibV1 : AHP\_depth\_last

Relative voltage values at the last after-hyperpolarization

- **Required features:** LibV5:voltage\_base (mV), LibV5:last\_AHP\_values (mV)
- **Units:** mV
- **Pseudocode:**

```
last_AHP_values = last_min_element(voltage, peak_indices)
AHP_depth = last_AHP_values[:] - voltage_base
```

#### LibV5 : AHP\_time\_from\_peak\_last

Time between AP peaks and last AHP depths

- **Required features:** LibV1:peak\_indices, LibV5:min\_AHP\_values (mV)
- **Units:** ms
- **Pseudocode:**

```
last_AHP_indices = last_min_element(voltage, peak_indices)
AHP_time_from_peak_last = t[last_AHP_indices[:]] - t[peak_indices[i]]
```

### LibV5 : steady\_state\_voltage\_stimend\_from\_voltage\_base

The average voltage during the last 90% of the stimulus duration relative to voltage\_base

- **Required features:** LibV5: steady\_state\_voltage\_stimend (mV), LibV5: voltage\_base (mV)
- **Units:** mV
- **Pseudocode:**

```
steady_state_voltage_stimend_from_voltage_base = steady_state_voltage_stimend -  
↔voltage_base
```

### LibV5 : min\_duringstim\_from\_voltage\_base

The minimum voltage during stimulus

- **Required features:** LibV5: min\_duringstim (mV), LibV5: voltage\_base (mV)
- **Units:** mV
- **Pseudocode:**

```
min_duringstim_from_voltage_base = minimum_voltage - voltage_base
```

### LibV5 : max\_duringstim\_from\_voltage\_base

The minimum voltage during stimulus

- **Required features:** LibV5: max\_duringstim (mV), LibV5: voltage\_base (mV)
- **Units:** mV
- **Pseudocode:**

```
max_duringstim_from_voltage_base = maximum_voltage - voltage_base
```

### LibV5 : diff\_max\_duringstim

Difference between maximum and steady state during stimulation

- **Required features:** LibV5: max\_duringstim (mV), LibV5: steady\_state\_voltage\_stimend (mV)
- **Units:** mV
- **Pseudocode:**

```
diff_max_duringstim: max_duringstim - steady_state_voltage_stimend
```



### LibV5 : diff\_min\_duringstim

Difference between minimum and steady state during stimulation

- **Required features:** LibV5: min\_duringstim (mV), LibV5: steady\_state\_voltage\_stimend (mV)
- **Units:** mV
- **Pseudocode:**

```
diff_min_duringstim: min_duringstim - steady_state_voltage_stimend
```

## 3.2.2 Python features

### Python efeature : initburst\_sahp

Slow AHP voltage after initial burst

The end of the initial burst is detected when the ISIs frequency gets lower than initburst\_freq\_threshold, in Hz. Then the sahp is searched for the interval between initburst\_sahp\_start (in ms) after the last spike of the burst, and initburst\_sahp\_end (in ms) after the last spike of the burst.

- **Required features:** LibV1: peak\_time
- **Parameters:** initburst\_freq\_threshold (default=50), initburst\_sahp\_start (default=5), initburst\_sahp\_end (default=100)
- **Units:** mV

### Python efeature : initburst\_sahp\_ssse

Slow AHP voltage from steady\_state\_voltage\_stimend after initial burst

- **Required features:** LibV5: steady\_state\_voltage\_stimend, initburst\_sahp
- **Units:** mV
- **Pseudocode:**

```
numpy.array([initburst_sahp_value[0] - ssse[0]])
```

### Python efeature : initburst\_sahp\_vb

Slow AHP voltage from voltage base after initial burst

- **Required features:** LibV5: voltage\_base, initburst\_sahp
- **Units:** mV
- **Pseudocode:**

```
numpy.array([initburst_sahp_value[0] - voltage_base[0]])
```

### Python efeature : depol\_block\_bool

Check for a depolarization block. Returns 1 if there is a depolarization block or a hyperpolarization block, and returns 0 otherwise.

A depolarization block is detected when the voltage stays higher than the mean of AP\_begin\_voltage for longer than 50 ms.

A hyperpolarization block is detected when, after stimulus start, the voltage stays below -75 mV for longer than 50 ms.

- **Required features:** LibV5: AP\_begin\_voltage
- **Units:** constant

### Python efeature : spikes\_per\_burst

Number of spikes in each burst.

The first spike is ignored by default. This can be changed by setting ignore\_first\_ISI to 0.

The burst detection can be fine-tuned by changing the setting strict\_burst\_factor. Default value is 2.0.

- **Required features:** LibV5: burst\_begin\_indices, LibV5: burst\_end\_indices
- **Units:** constant
- **Pseudocode:**

```
spike_per_bursts = []
for idx_begin, idx_end in zip(burst_begin_indices, burst_end_indices):
    spike_per_bursts.append(idx_end - idx_begin + 1)
```

### Python efeature : spikes\_per\_burst\_diff

Difference of number of spikes between each burst and the next one.

The first spike is ignored by default. This can be changed by setting ignore\_first\_ISI to 0.

The burst detection can be fine-tuned by changing the setting strict\_burst\_factor. Default value is 2.0.

- **Required features:** spikes\_per\_burst
- **Units:** constant
- **Pseudocode:**

```
spikes_per_burst[:-1] - spikes_per_burst[1:]
```

### Python efeature : spikes\_in\_burst1\_burst2\_diff

Difference of number of spikes between the first burst and the second one.

The first spike is ignored by default. This can be changed by setting ignore\_first\_ISI to 0.

The burst detection can be fine-tuned by changing the setting strict\_burst\_factor. Default value is 2.0.

- **Required features:** spikes\_per\_burst\_diff
- **Units:** constant
- **Pseudocode:**

```
numpy.array([spikes_per_burst_diff[0]])
```

### Python efeature : spikes\_in\_burst1\_burstlast\_diff

Difference of number of spikes between the first burst and the last one.

The first spike is ignored by default. This can be changed by setting `ignore_first_ISI` to 0.

The burst detection can be fine-tuned by changing the setting `strict_burst_factor`. Default value is 2.0.

- **Required features:** `spikes_per_burst`
- **Units:** constant
- **Pseudocode:**

```
numpy.array([spikes_per_burst[0] - spikes_per_burst[-1]])
```

### Python efeature : impedance

Computes the impedance given a ZAP current input and its voltage response. It will return the frequency at which the impedance is maximal, in the range (0, `impedance_max_freq`] Hz, with `impedance_max_freq` being a setting with 50.0 as a default value.

- **Required features:** `current`, `LibV1:Spikecount`, `LibV5:voltage_base`, `LibV5:current_base`
- **Units:** Hz
- **Pseudocode:**

```
normalized_voltage = voltage_trace - voltage_base
normalized_current = current_trace - current_base
if spike_count < 1: # if there is no spikes in ZAP
    fft_volt = numpy.fft.fft(normalized_voltage)
    fft_cur = numpy.fft.fft(normalized_current)
    if any(fft_cur) == 0:
        return None
    # convert dt from ms to s to have freq in Hz
    freq = numpy.fft.fftfreq(len(normalized_voltage), d=dt / 1000.)
    Z = fft_volt / fft_cur
    norm_Z = abs(Z) / max(abs(Z))
    select_idxs = numpy.swapaxes(numpy.argwhere((freq > 0) & (freq <= impedance_max_
↪freq)), 0, 1)[0]
    smooth_Z = gaussian_filter1d(norm_Z[select_idxs], 10)
    ind_max = numpy.argmax(smooth_Z)
    return freq[ind_max]
else:
    return None
```



## PYTHON API

eFEL Python API functions.

This module provides the user-facing Python API of the eFEL. The convenience functions defined here call the underlying ‘cppcore’ library to hide the lower level API from the user. All functions in this module can be called as `efel.functionname`, it is not necessary to include ‘api’ as in `efel.api.functionname`.

Copyright (c) 2015, EPFL/Blue Brain Project

This file is part of eFEL <<https://github.com/BlueBrain/eFEL>>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License version 3.0 as published by the Free Software Foundation.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

`efel.api.FeatureNameExists(feature_name)`

Does a certain feature name exist ?

**Parameters**

**feature\_name** (*string*) – Name of the feature to check

**Returns**

**FeatureNameExists** – True if feature\_name exists, otherwise False

**Return type**

bool

`efel.api.getDependencyFileLocation()`

Get the location of the Dependency file

The eFEL uses ‘Dependency’ files to let the user define which versions of certain features are used to calculate. The installation directory of the eFEL contains a default ‘DependencyV5.txt’ file.

**Returns**

**location** – path to the location of a Dependency file

**Return type**

string

`efel.api.getDistance(trace, featureName, mean, std, trace_check=True, error_dist=250)`

Calculate distance value for a list of traces.

**Parameters**

- **trace** (*trace dicts*) – Trace dict that represents one trace. The dict should have the following keys: ‘T’, ‘V’, ‘stim\_start’, ‘stim\_end’
- **featureName** (*string*) – Name of the the features for which to calculate the distance
- **mean** (*float*) – Mean to calculate the distance from
- **std** (*float*) – Std to scale the distance with
- **trace\_check** (*float*) – Let the library check if there are spikes outside of stimulus interval, default is True
- **error\_dist** (*float*) – Distance returned when error, default is 250

**Returns**

**distance** – The absolute number of standard deviation the feature is away from the mean. In case of anomalous results a value of ‘error\_dist’ standard deviations is returned. This can happen if: a feature generates an error, there are spikes outside of the stimulus interval, the feature returns a NaN, etc.

**Return type**

float

`efel.api.getFeatureNames()`

Return a list with the name of all the available features

**Returns**

**feature\_names** – A list that contains all the feature names available in the eFEL. These names can be used in the featureNames argument of e.g. getFeatureValues()

**Return type**

list of strings

`efel.api.getFeatureValues(traces, featureNames, parallel_map=None, return_list=True, raise_warnings=True)`

Calculate feature values for a list of traces.

This function is the core of the eFEL API. A list of traces (in the form of dictionaries) is passed as argument, together with a list of feature names.

The return value consists of a list of dictionaries, one for each input trace. The keys in the dictionaries are the names of the calculated features, the corresponding values are lists with the feature values. Beware that every feature returns an array of values. E.g. AP\_amplitude will return a list with the amplitude of every action potential.

**Parameters**

- **traces** (*list of trace dicts*) – Every trace dict represent one trace. The dict should have the following keys: ‘T’, ‘V’, ‘stim\_start’, ‘stim\_end’
- **feature\_names** (*list of string*) – List with the names of the features to be calculated on all the traces.
- **parallel\_map** (*map function*) – Map function to parallelise over the traces. Default is the serial map() function
- **return\_list** (*boolean*) – By default the function returns a list of dicts. This optional argument can disable this, so that the result of the parallel\_map() is returned. Can be useful for performance reasons when an iterator is preferred.
- **raise\_warnings** (*boolean*) – Raise warning when efel c++ returns an error

**Returns**

**feature\_values** – For every input trace a feature value dict is return (in the same order). The dict contains the keys of ‘feature\_names’, every key contains a numpy array with the feature values returned by the C++ efel code. The value is None if an error occured during the calculation of the feature.

**Return type**

list of dicts

`efel.api.getMeanFeatureValues(traces, featureNames, raise_warnings=True)`

Convenience function that returns mean values from getFeatureValues()

Instead of return a list of values for every feature as getFeatureValues() does, this function returns per trace one value for every feature, namely the mean value.

**Parameters**

- **traces** (*list of trace dicts*) – Every trace dict represent one trace. The dict should have the following keys: ‘T’, ‘V’, ‘stim\_start’, ‘stim\_end’
- **feature\_names** (*list of string*) – List with the names of the features to be calculated on all the traces.
- **raise\_warnings** (*boolean*) – Raise warning when efel c++ returns an error

**Returns**

**feature\_values** – For every input trace a feature value dict is return (in the same order). The dict contains the keys of ‘feature\_names’, every key contains the mean of the array that is returned by getFeatureValues() The value is None if an error occured during the calculation of the feature, or if the feature value array was empty.

**Return type**

list of dicts

`efel.api.get_cpp_feature(featureName, raise_warnings=None)`

Return value of feature implemented in cpp

`efel.api.get_py_feature(featureName)`

Return python feature

`efel.api.reset()`

Resets the efel to its initial state

The user can set certain values in the efel, like the spike threshold. These values are persisten. This function will reset these value to their original state.

`efel.api.setDependencyFileLocation(location)`

Set the location of the Dependency file

The eFEL uses ‘Dependency’ files to let the user define which versions of certain features are used to calculate. The installation directory of the eFEL contains a default ‘DependencyV5.txt’ file. Unless the user wants to change this file, it is not necessary to call this function.

**Parameters**

**location** (*string*) – path to the location of a Dependency file

`efel.api.setDerivativeThreshold(newDerivativeThreshold)`

Set the threshold for the derivate for detecting the spike onset

Some featurea use a threshold on dV/dt to calculate the beginning of an action potential. This function allows you to set this threshold.

**Parameters**

**derivative\_threshold** (*float*) – The new derivative threshold value (in the same units as the traces, e.g. mV/ms).

`efel.api.setDoubleSetting(setting_name, new_value)`

Set a certain double setting to a new value

`efel.api.setIntSetting(setting_name, new_value)`

Set a certain integer setting to a new value

`efel.api.setStrSetting(setting_name, new_value)`

Set a certain string setting to a new value

`efel.api.setThreshold(newThreshold)`

Set the spike detection threshold in the eFEL, default -20.0

**Parameters**

**threshold** (*float*) – The new spike detection threshold value (in the same units as the traces, e.g. mV).



## DEVELOPER'S GUIDE

### Contents

- *Developer's Guide*
  - *Requirements*
  - *Forking and cloning the git repository*
  - *Makefile*
  - *Adding a new eFeature*
    - \* *Picking a name*
    - \* *Creating a branch*
    - \* *Implementation*
    - \* *Updating relevant files*
    - \* *Adding a test*
    - \* *Add documentation*
    - \* *Pull request*

## 5.1 Requirements

As a developer you will need some extra requirements

- To get the latest source code: [Git](#)
- To run the tests: [Pytest](#)
- To build the documentation: [Sphinx](#), and `pdflatex` (e.g. from [Mactex](#))

## 5.2 Forking and cloning the git repository

To make changes to the eFEL, one first needs to fork the eFEL:

```
https://help.github.com/articles/fork-a-repo/
```

Then one creates a local clone of the git repository on your computer:

```
git clone https://github.com/yourgithubusername/eFEL.git
```

After changes are made, they should be pushed back to your github account. Then a pull request can be created:

```
https://help.github.com/articles/using-pull-requests/
```

## 5.3 Makefile

To simplify certain tasks for developers, a Makefile is provided in the root of the eFEL project. This Makefile has the following targets

- **install**: installs the eFEL using pip from the working directory
- **test**: run the installation and all the tests
- **doc**: build the sphinx and latex documentation
- **clean**: clean up the build directories
- **pypi**: run test target and upload to pypi
- **push**: clean the build, update the version from the git hash, install eFEL, run the tests, build the doc, and push the documentation and source to github

## 5.4 Adding a new eFeature

Adding a new eFeature requires several steps.

### 5.4.1 Picking a name

Try to be specific in the name of the eFeature, because in the future you or somebody else might want to develop an eFeature with slightly different behavior. Don't be afraid to use long names, e.g. 'min\_voltage\_between\_spikes' is perfectly ok.

## 5.4.2 Creating a branch

Create a git branch with the name of the new eFeature:

```
git checkout -b your_efeaturename
```

## 5.4.3 Implementation

All the eFeatures in the eFEL are coded in C++. Thanks to an [eFeatures dependency settings file](#), several implementation of the same eFeature name can coexist. E.g. [this](#) is the file with the implementations of all 'V5' features. You can implement the new eFeature by extending one of the current LibV\* files, or by creating your own. You might want to consider starting the implementation by writing a test for the eFeature (see below for instruction on how to do that).

## 5.4.4 Updating relevant files

Apart from the implementation in the LibV\*.cpp file, other files have to be changed to accomodate the new eFeature

- efel/cppcore/LibV5.h: Declare your feature
- efel/DependencyV5.txt: Add your eFeature and its dependencies to this file
- efel/cppcore/FillFptrTable.cpp: Add a reference to the eFeature in the relevant table
- efel/cppcore/cfeature.cpp: Add the type of the eFeature
- AUTHORS.txt: If your name isn't there yet, add yourself to the authors list
- efel/units/units.json: Add the units of the eFeature to the API

You can confirm everything compiles correctly by executing:

```
make test
```

## 5.4.5 Adding a test

Most eFeatures are fairly easy to implement in Python, so it is advised to first write a Python implementation of your eFeature, and to add a test to it. Then, while you are implementing the code in C++ you can easily compare the results to the test.

The tests of the individual eFeatures are [here](#). Just add your own test by defining a new function 'test\_yourfeature()'.  
Some test data is available [at this link](#), but you can of course add your own traces.

The easiest way to run the tests is by executing:

```
make test
```

## 5.4.6 Add documentation

Add the documentation of the new eFeature to this file:

<https://github.com/BlueBrain/eFEL/blob/master/docs/source/eFeatures.rst>

Please provide some pseudo-Python code for the eFeature.

The documentation can be built by:

```
make doc
```

It can be viewed by opening:

```
docs/build/html/index.html
```

To build the documentation, pdflatex has to be present on the system. On a Mac this can be installed using [Mactex](#). On Ubuntu one can use:

```
sudo apt-get install texlive-latex-base texlive-latex-extra xzdec  
tlmgr install helvetic
```

## 5.4.7 Pull request

When all the above steps were succesfull, you can push the new eFeature branch to your github repository:

```
git commit -a  
git push origin your_efeaturename
```

Finally create a pull request:

<https://help.github.com/articles/using-pull-requests/>

## INDICES AND TABLES

- genindex
- modindex
- search



## PYTHON MODULE INDEX

**e**

`efel.api`, 57





## E

`efel.api`  
module, 57

## F

`FeatureNameExists()` (in module *efel.api*), 57

## G

`get_cpp_feature()` (in module *efel.api*), 59  
`get_py_feature()` (in module *efel.api*), 59  
`getDependencyFileLocation()` (in module *efel.api*),  
57  
`getDistance()` (in module *efel.api*), 57  
`getFeatureNames()` (in module *efel.api*), 58  
`getFeatureValues()` (in module *efel.api*), 58  
`getMeanFeatureValues()` (in module *efel.api*), 59

## M

module  
`efel.api`, 57

## R

`reset()` (in module *efel.api*), 59

## S

`setDependencyFileLocation()` (in module *efel.api*),  
59  
`setDerivativeThreshold()` (in module *efel.api*), 59  
`setDoubleSetting()` (in module *efel.api*), 60  
`setIntSetting()` (in module *efel.api*), 60  
`setStrSetting()` (in module *efel.api*), 60  
`setThreshold()` (in module *efel.api*), 60