eFEL Documentation

Release 5.6.11+1.g69f1c1b

BBP, EPFL

Apr 26, 2024

CONTENTS

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. You can automatically compile and install the library as a Python module.

The source code of the eFEL is located on github: BlueBrain/eFEL

CHAPTER

ONE

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
```

CHAPTER

TWO

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.get_feature_names()
```

The python function to extract eFeatures is get_feature_values(...). 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.get_feature_values(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

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)

(continued from previous 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.get_feature_values(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 you 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.get_feature_values(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")
```

2.1.4 Example of the use of the eFEL (eFeature Extraction Library) in conjunction with models downloaded from the Neocortical Microcircuit Portal website

Requirements: - Python 3.9+, including Pip (https://pip.readthedocs.org) - A version of Neuron (with Python support) installed on your computer (for instruction, see https://bbp.epfl.ch/nmc-portal/tools)

Make matplotlib plots show up in the notebook:

```
%matplotlib inline
```

Install the eFeature Extraction Library:

```
!pip install efel
import efel
```

Get a model package from the website

```
!curl -o L5_TTPC2.zip https://bbp.epfl.ch/nmc-portal/assets/documents/static/downloads-
↪zip/L5_TTPC2_cADpyr232_1.zip
```

Unzip the model package:

```
!unzip L5_TTPC2.zip
```

Change directory to the model package directory:

```
import os
os.chdir('L5_TTPC2_cADpyr232_1')
```

Compile the Neuron mechanisms (if this fails, you might not have installed Neuron correctly)

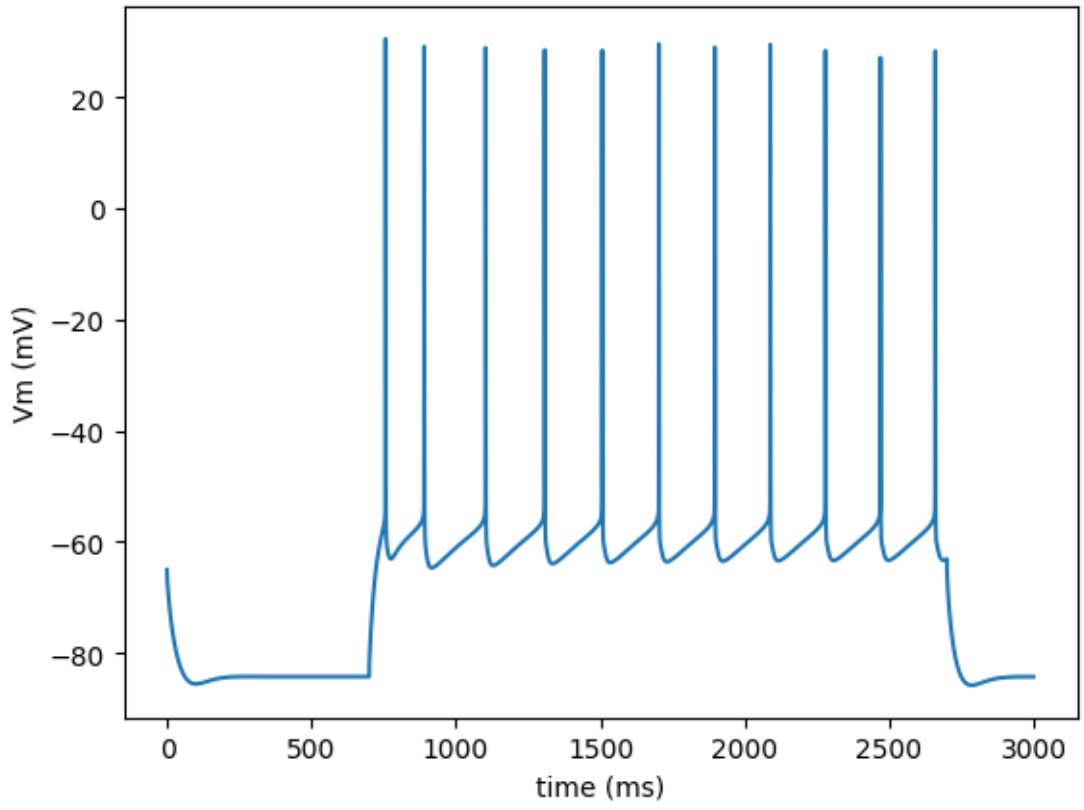
```
!nrnivmodl ./mechanisms
```

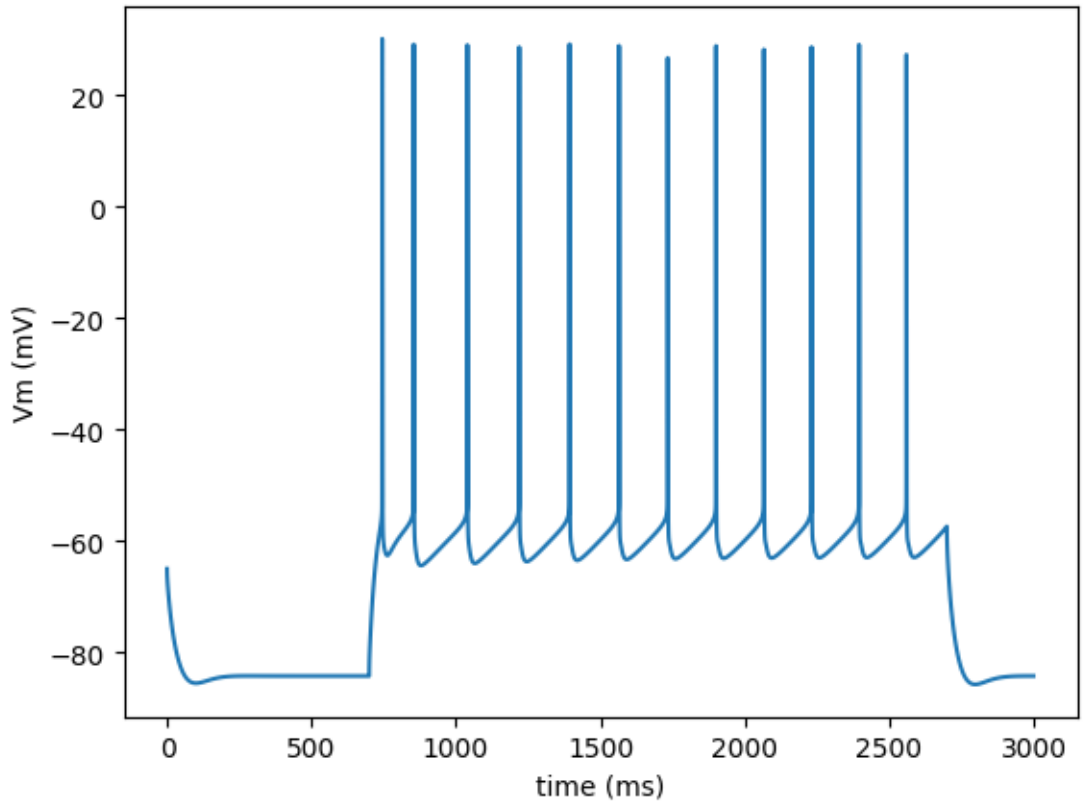
Import the model package in Python, and run it:

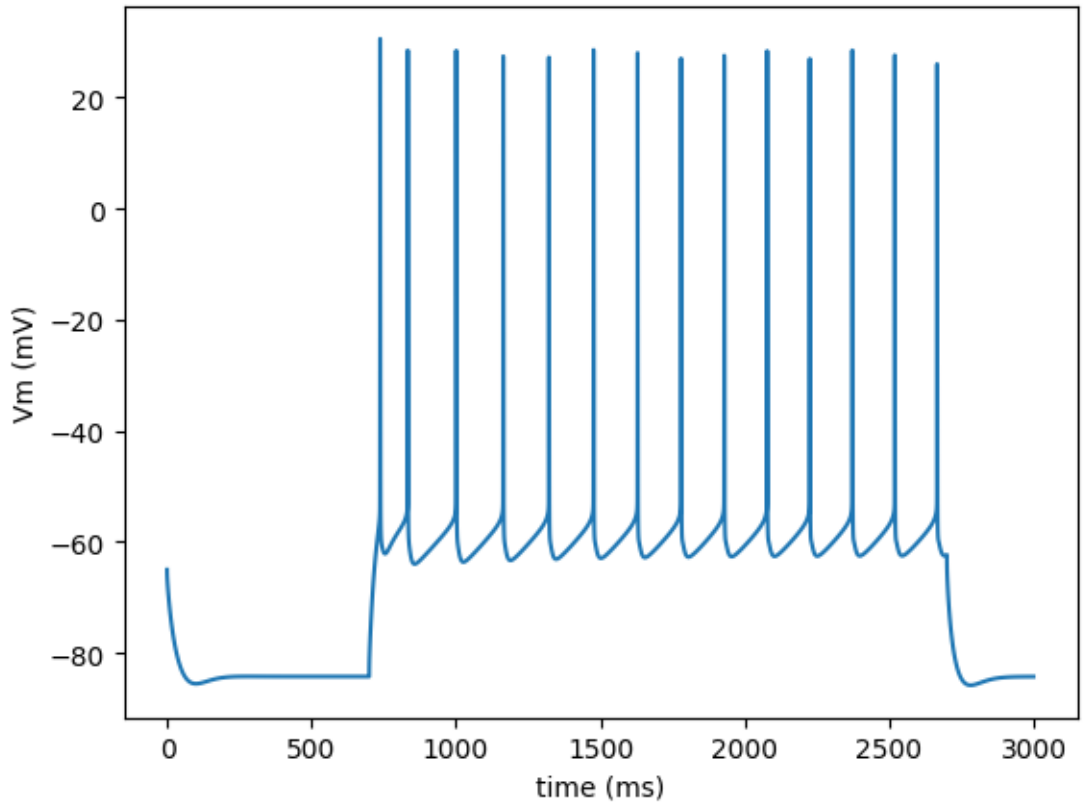
```
import run
run.main(plot_traces=True)
```

```
Warning: no DISPLAY environment variable.
--No graphics will be displayed.
```

```
Loading constants
Setting temperature to 34.000000 C
Setting simulation time step to 0.025000 ms
    1
    1
    1
Loading cell cADpyr232_L5_TTPC2_8052133265
Attaching stimulus electrodes
Setting up step current clamp: amp=0.593063 nA, delay=700.000000 ms, duration=2000.
↪000000 ms
Setting up hypamp current clamp: amp=-0.286011 nA, delay=0.000000 ms, duration=3000.
↪000000 ms
Attaching recording electrodes
Setting simulation time to 3s for the step currents
Disabling variable timestep integration
Running for 3000.000000 ms
Soma voltage for step 1 saved to: python_recordings/soma_voltage_step1.dat
Loading cell cADpyr232_L5_TTPC2_8052133265
Attaching stimulus electrodes
Setting up step current clamp: amp=0.642485 nA, delay=700.000000 ms, duration=2000.
↪000000 ms
Setting up hypamp current clamp: amp=-0.286011 nA, delay=0.000000 ms, duration=3000.
↪000000 ms
Attaching recording electrodes
Setting simulation time to 3s for the step currents
Disabling variable timestep integration
Running for 3000.000000 ms
Soma voltage for step 2 saved to: python_recordings/soma_voltage_step2.dat
Loading cell cADpyr232_L5_TTPC2_8052133265
Attaching stimulus electrodes
Setting up step current clamp: amp=0.691907 nA, delay=700.000000 ms, duration=2000.
↪000000 ms
Setting up hypamp current clamp: amp=-0.286011 nA, delay=0.000000 ms, duration=3000.
↪000000 ms
Attaching recording electrodes
Setting simulation time to 3s for the step currents
Disabling variable timestep integration
Running for 3000.000000 ms
Soma voltage for step 3 saved to: python_recordings/soma_voltage_step3.dat
```


20
0
−20
−40
−60
−80
Vm (mV)
0
500
1000
1500
2000
2500
3000
time (ms)


20
0
−20
−40
−60
−80
Vm (mV)
0
500
1000
1500
2000
2500
3000
time (ms)



Load the output of the model package in numpy array

```
import numpy
times = []
voltages = []
for step_number in range(1,4):
    data = numpy.loadtxt('python_recordings/soma_voltage_step%d.dat' % step_number)
    times.append(data[:, 0])
    voltages.append(data[:, 1])
```

Prepare the traces for the eFEL

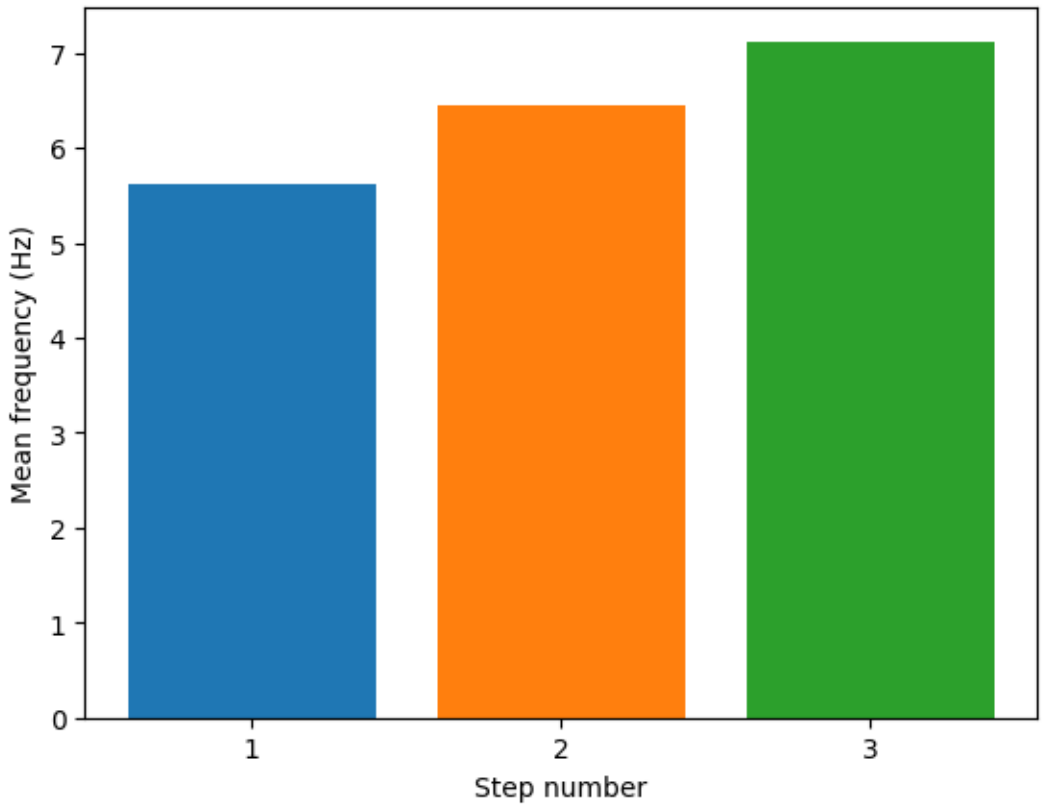
```
traces = []
for step_number in range(3):
    trace = {}
    trace['T'] = times[step_number]
    trace['V'] = voltages[step_number]
    trace['stim_start'] = [700]
    trace['stim_end'] = [2700]
    traces.append(trace)
```

Run the eFEL on the trace

```
feature_values = efel.get_feature_values(traces, ['mean_frequency', 'adaptation_index2',
↪'ISI_CV', 'doublet_ISI', 'time_to_first_spike', 'AP_height', 'AHP_depth_abs', 'AHP_
↪depth_abs_slow', 'AHP_slow_time', 'AP_width', 'peak_time', 'AHP_time_from_peak'])
```

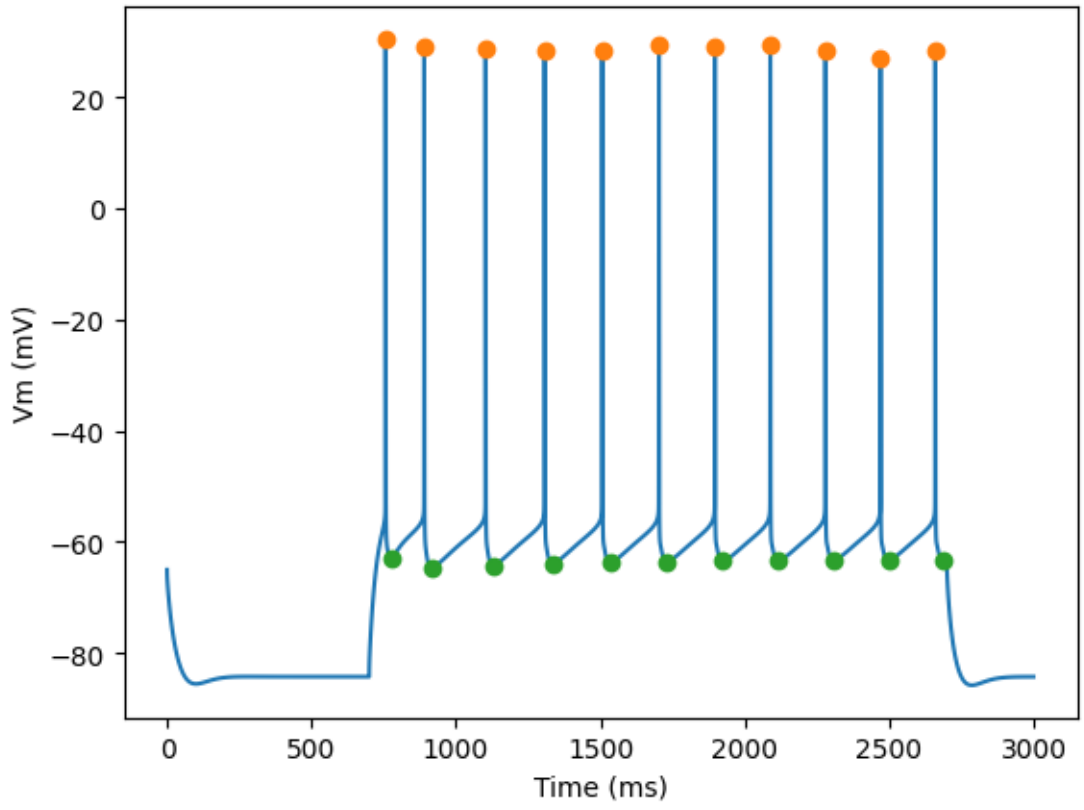
Plot frequencies over steps

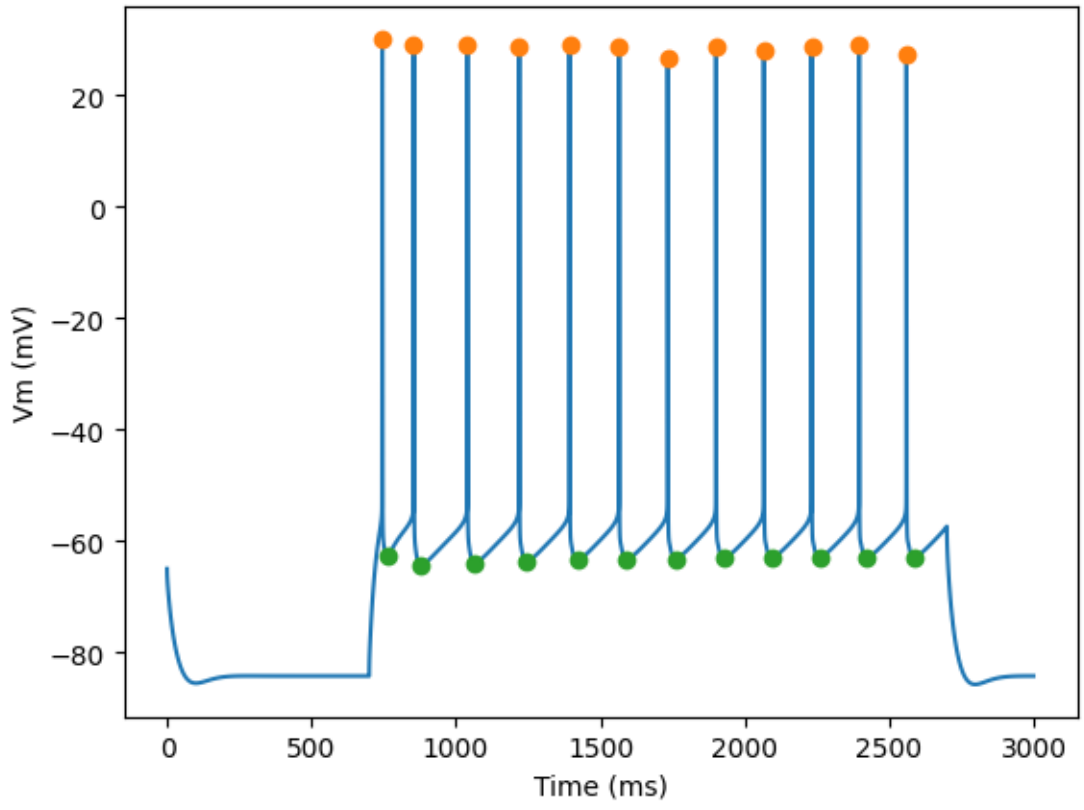
```
import pylab
for step_number in range(3):
    pylab.bar(step_number, feature_values[step_number]['mean_frequency'][0], align=
↪'center')
pylab.ylabel('Mean frequency (Hz)')
pylab.xlabel('Step number')
pylab.xticks(range(3), range(1,4))
pylab.show()
```

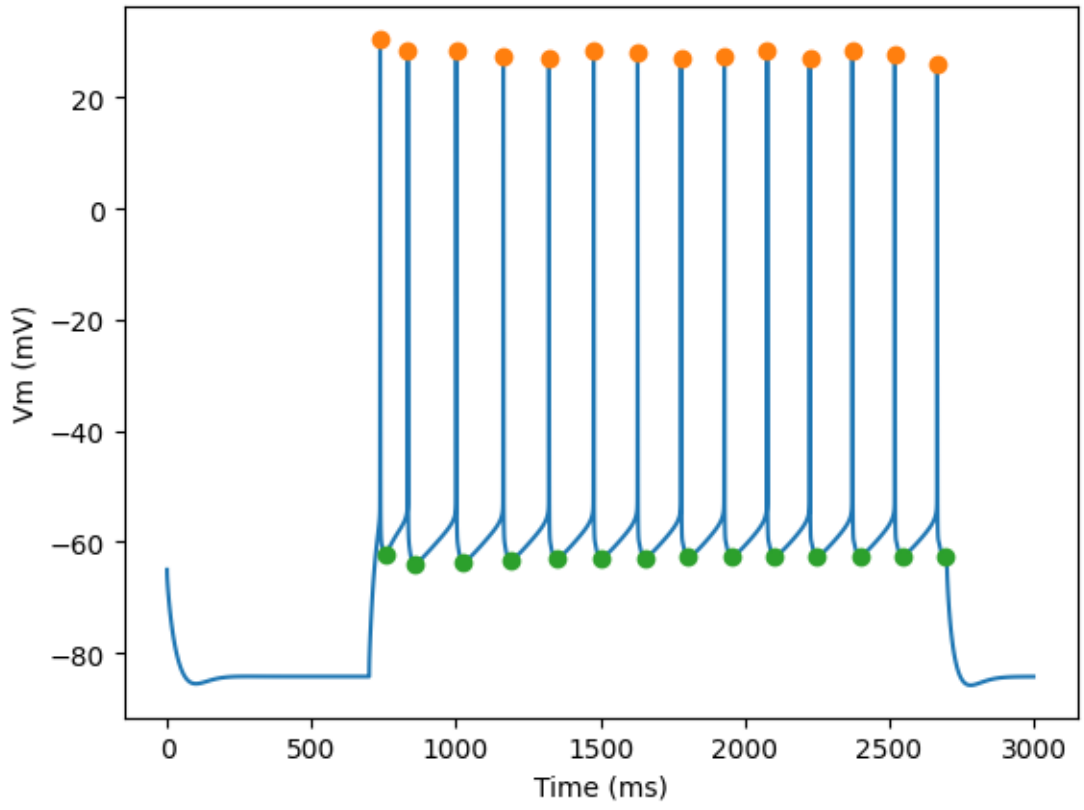


Plot AP height and AHP depth

```
for step_number in range(3):
    time = times[step_number]
    voltage = voltages[step_number]
    peak_times = feature_values[step_number]['peak_time']
    ahp_time = feature_values[step_number]['AHP_time_from_peak']
    ap_heights = feature_values[step_number]['AP_height']
    AHP_depth_abss = feature_values[step_number]['AHP_depth_abs']

    pylab.plot(time,voltage)
    pylab.plot(peak_times, ap_heights, 'o')
    pylab.plot(peak_times+ahp_time, AHP_depth_abss, 'o')
    pylab.xlabel('Time (ms)')
    pylab.ylabel('Vm (mV)')
    pylab.show()
```


20
0
−20
−40
−60
−80
Vm (mV)
0
500
1000
1500
2000
2500
3000
Time (ms)


20
0
−20
−40
−60
−80
Vm (mV)
0
500
1000
1500
2000
2500
3000
Time (ms)



2.1.5 Extracting features from SONATA Network simulations

This notebook shows how to extract features of a group of cells from a SONATA network, specifically focusing on a small portion of non-barrel primary somatosensory cortex circuit from juvenile rats, with the help of BlueCellulLab. For those interested in conducting a more in-depth analysis, the entire circuit dataset is accessible on Zenodo. For more details about the simulation and in-depth insights on the circuit, please refer to the Bluecellulab SONATA Network example and the related paper, respectively.

Note: The compiled mechanisms need to be provided before importing bluecellulab.

```
!nrnivmodl ./mechanisms
```

```
import json
from pathlib import Path

from matplotlib import pyplot as plt

from bluecellulab import CircuitSimulation
import efel
```

In this example, a small sub-circuit has been extracted from the sscx circuit. This sub-circuit specifically consists of a random selection of cells exhibiting delayed stuttering (dSTUT) etype.

The simulation_config specifies the types of input stimuli to be applied to the cells. In this case, we have selected a 'relative_linear' stimulus of 70 ms and set the stimulus current at a level equivalent to 100 percent of the cell's threshold current.

```
simulation_config = Path("./") / "simulation_config.json"
with open(simulation_config) as f:
    simulation_config_dict = json.load(f)
print(json.dumps(simulation_config_dict, indent=4))
```


```
{
    "manifest": {
        "$OUTPUT_DIR": "."
    },
    "run": {
        "tstop": 100.0,
        "dt": 0.025,
        "random_seed": 1
    },
    "conditions": {
        "v_init": -65
    },
    "target_simulator": "NEURON",
    "network": "./O1/circuit_config.json",
    "node_set": "dSTUT_mini",
    "output": {
        "output_dir": "$OUTPUT_DIR/output_sonata",
        "spikes_file": "out.h5",
        "spikes_sort_order": "by_time"
    },
    "inputs": {
        "continuous_linear": {
            "input_type": "current_clamp",
            "module": "relative_linear",
            "delay": 20.0,
            "duration": 70.0,
            "percent_start": 100,
            "node_set": "dSTUT_mini"
        }
    },
    "reports": {
        "soma": {
            "cells": "dSTUT_mini",
            "variable_name": "v",
            "type": "compartment",
            "dt": 1.0,
            "start_time": 0.0,
            "end_time": 20.0,
            "sections": "soma",
            "compartments": "center"
        }
    }
}
```


We use BlueCellulab for simulating smaller scale circuits, in contrast to the larger-scale simulations conducted with Neurodamus.

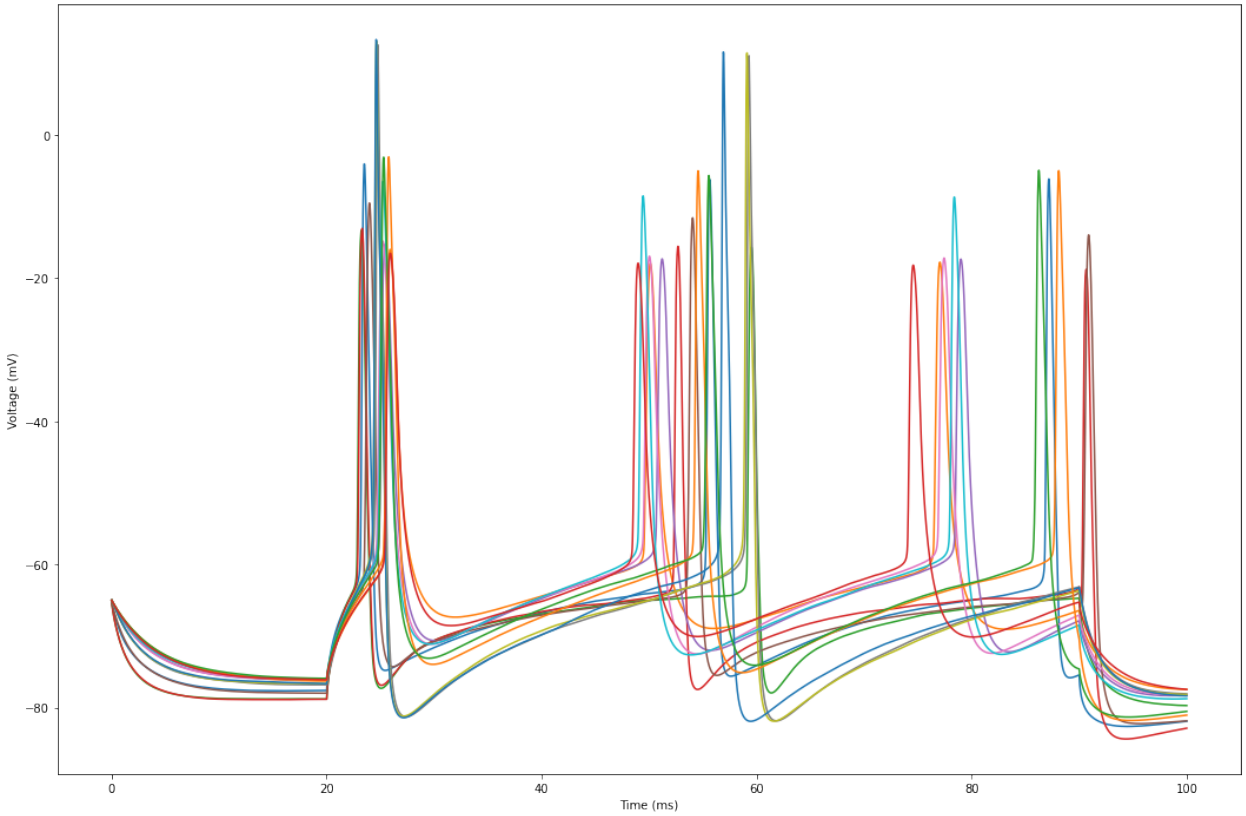
```
simulation_config = Path("./") / "simulation_config.json"
with open(simulation_config) as f:
    simulation_config_dict = json.load(f)
sim = CircuitSimulation(simulation_config)
```

```
from bluepysnap import Simulation as snap_sim
snap_access = snap_sim(simulation_config)
import pandas as pd
from bluepysnap import Simulation as snap_sim
all_nodes = pd.concat([x[1] for x in snap_access.circuit.nodes.get()])
dstut_cells = all_nodes[all_nodes["etype"] == "dSTUT"].index.to_list()
```

```
sim.instantiate_gids(dstut_cells, add_stimuli=True)
t_stop = 100.0
sim.run(t_stop)
```

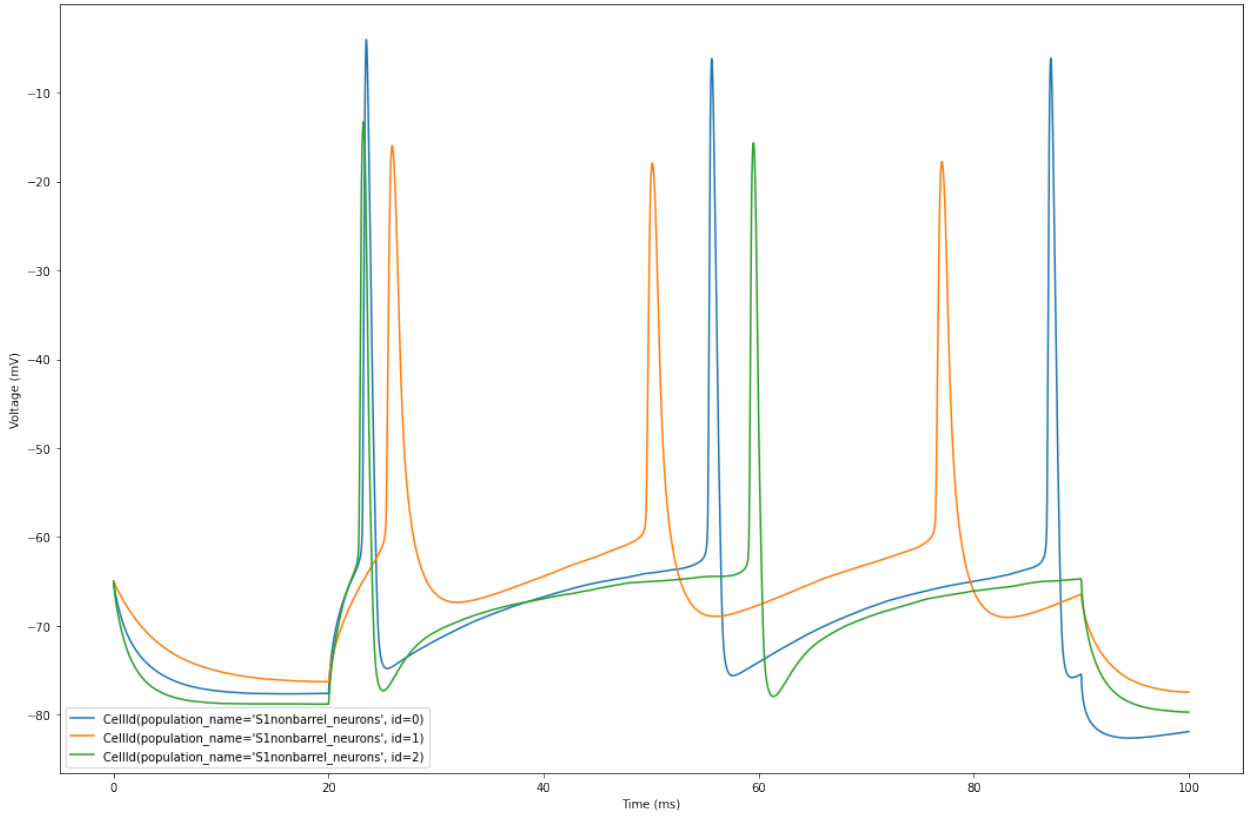
The plot displays the voltage traces simulated for each cell in our circuit.

```
plt.figure(figsize=(18, 12))
for cell_id in sim.cells:
    time = sim.get_time_trace()
    voltage = sim.get_voltage_trace(cell_id)
    plt.plot(time, voltage, label=cell_id)
    plt.xlabel("Time (ms)")
    plt.ylabel("Voltage (mV)")
```



Let's focus on 3 cells for better visualization

```
sim.cells = dict(list(sim.cells.items())[:3])
plt.figure(figsize=(18, 12))
for cell_id in sim.cells:
    time = sim.get_time_trace()
    voltage = sim.get_voltage_trace(cell_id)
    plt.plot(time, voltage, label=cell_id)
    plt.xlabel("Time (ms)")
    plt.ylabel("Voltage (mV)")
    plt.legend()
```



We are now ready to extract features. First, we will build the data structure for eFEL

```
traces = []
for cell_id in sim.cells:
    voltage = sim.get_voltage_trace(cell_id)
    trace = {}
    trace['T'] = time
    trace['V'] = voltage
    trace['stim_start'] = [20]
    trace['stim_end'] = [90]
    traces.append(trace)
```

Next, we choose the specific features of interest

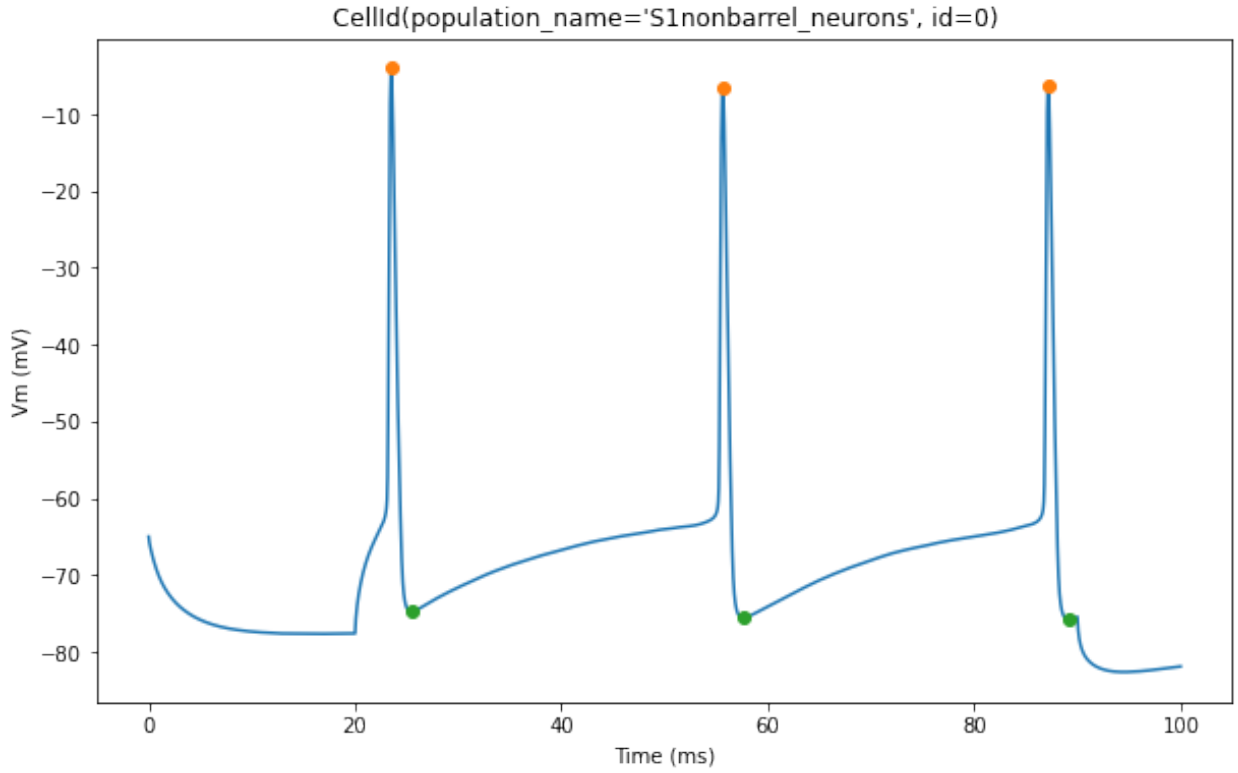
```
features = ['peak_time', 'AHP_time_from_peak', 'AP_height', 'AHP_depth_abs', 'all_ISI_
↪values']
```

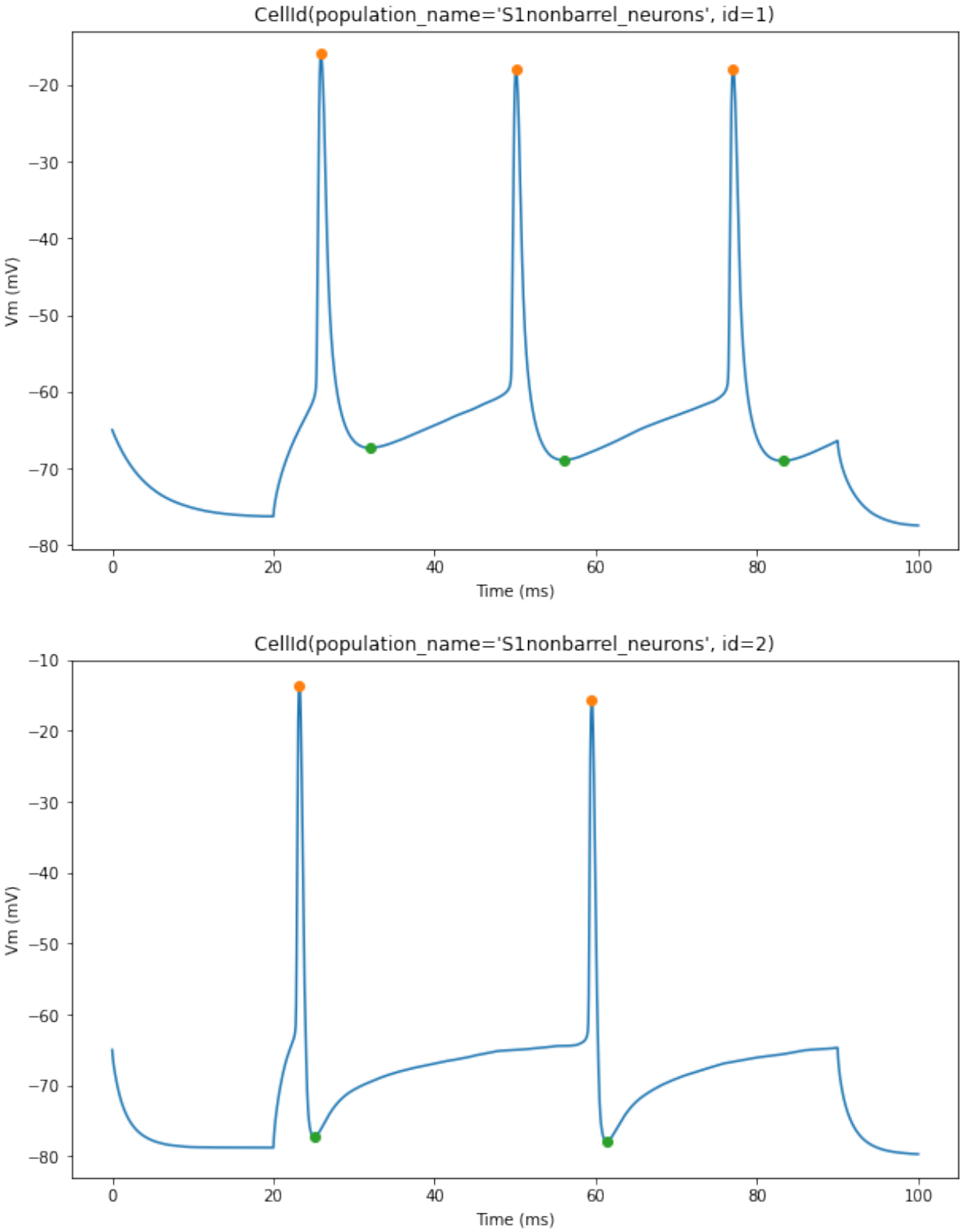
Finally, we perform the feature extraction

```
traces_results = efel.get_feature_values(traces, features)
```

The plot below shows action potential (AP) height and depth of those 3 cells

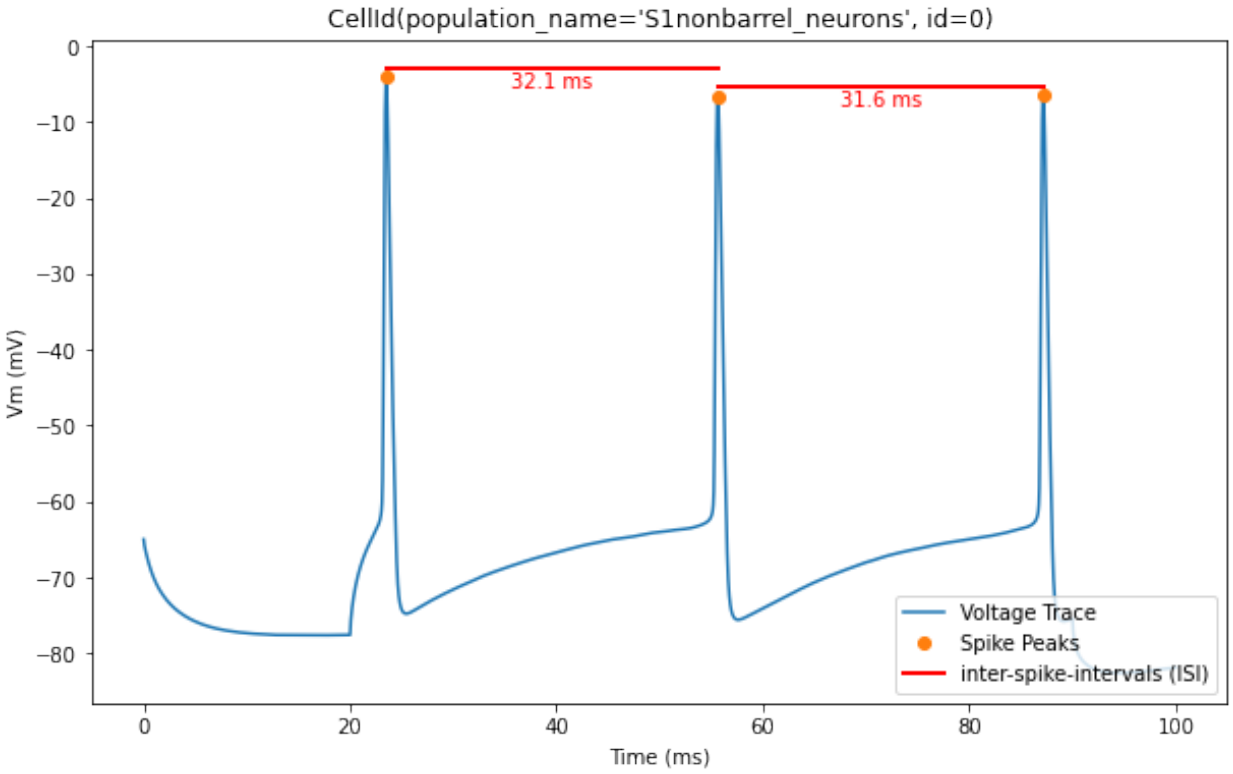
```
import pylab
for trace, trace_result, cell_id in zip(traces, traces_results, sim.cells):
    time = trace['T']
    voltage = trace['V']
    peak_times = trace_result['peak_time']
    ahp_time = trace_result['AHP_time_from_peak']
    ap_heights = trace_result['AP_height']
    AHP_depth_abss = trace_result['AHP_depth_abs']

    pylab.figure(figsize=(10, 6))
    pylab.title(cell_id)
    pylab.plot(time,voltage)
    pylab.plot(peak_times, ap_heights, 'o')
    pylab.plot(peak_times+ahp_time, AHP_depth_abss, 'o')
    pylab.xlabel('Time (ms)')
    pylab.ylabel('Vm (mV)')
    pylab.show()
```

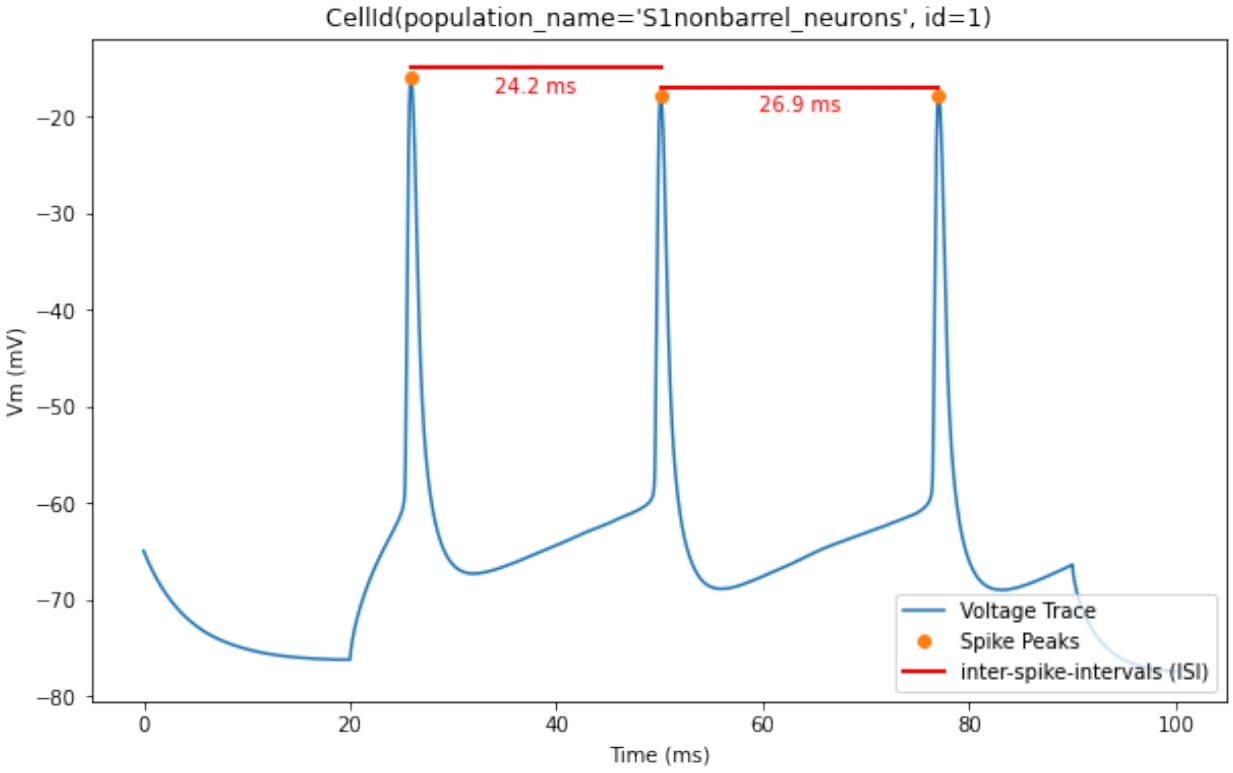


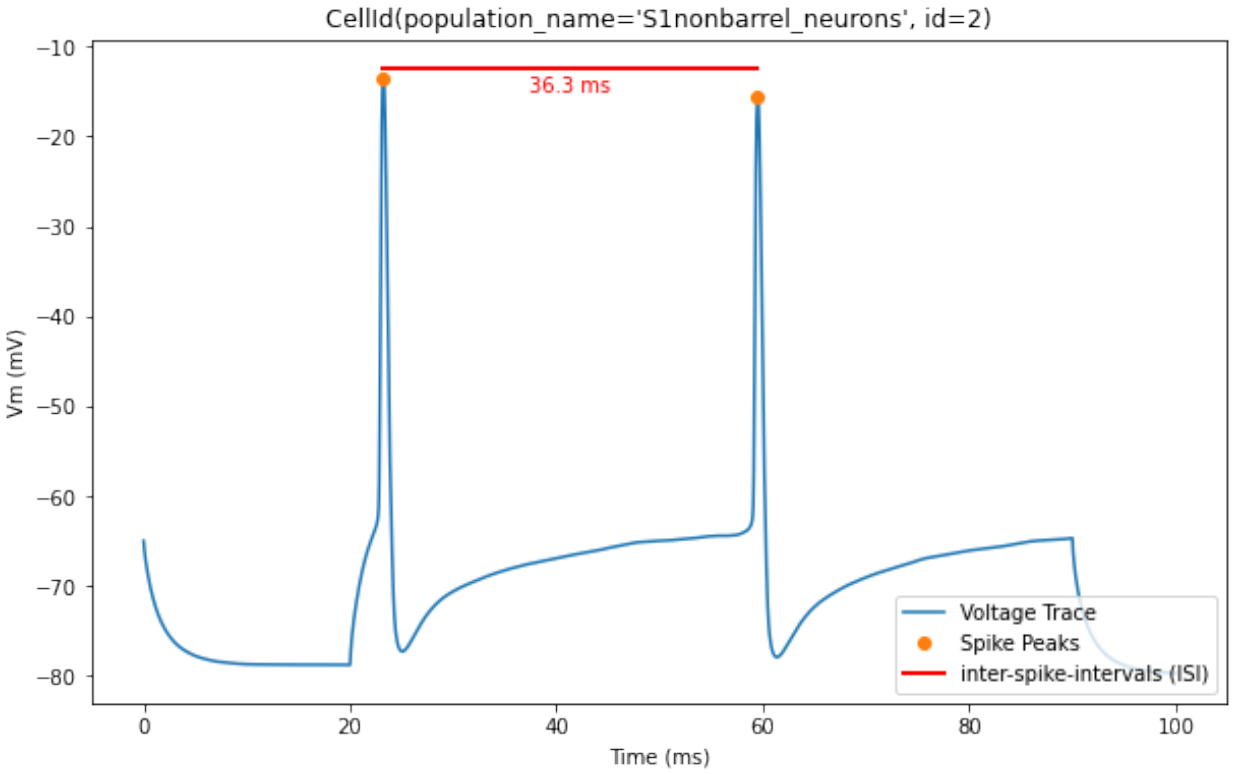


Now, let's overlay the durations of the inter-spike intervals (ISIs) for a clearer visualization of the timing between spikes

```
for trace, trace_result, cell_id in zip(traces, traces_results, sim.cells):
    time = trace['T']
    voltage = trace['V']
    peak_times = trace_result['peak_time']
    ap_heights = trace_result['AP_height']

    all_isi_values = trace_result['all_ISI_values']

    pylab.figure(figsize=(10, 6))
    pylab.title(cell_id)
    pylab.plot(time, voltage, label='Voltage Trace')
    pylab.plot(peak_times, ap_heights, 'o', label='Spike Peaks')

    for i in range(len(peak_times) - 1):
        start_spike_time = peak_times[i]
        end_spike_time = peak_times[i + 1]
        duration = round(all_isi_values[i], 2)

        y_position = max(ap_heights[i], ap_heights[i + 1]) + 1

        # Check if it's the first ISI line to be drawn and add a label, otherwise draw␣
↪without a label
        if i == 0:
            pylab.plot([start_spike_time, end_spike_time], [y_position, y_position], 'r-
↪', lw=2, label='inter-spike-intervals (ISI)')
        else:
            pylab.plot([start_spike_time, end_spike_time], [y_position, y_position], 'r-
↪', lw=2)

        # Adjust text position to be slightly lower
        midpoint = (start_spike_time + end_spike_time) / 2
        pylab.text(midpoint, y_position - 3, f'{duration} ms', verticalalignment='bottom
↪', horizontalalignment='center', color='red')

    pylab.xlabel('Time (ms)')
    pylab.ylabel('Vm (mV)')
    pylab.legend(loc='lower right')
    pylab.show()
```


CellId(population_name='S1nonbarrel_neurons', id=0)
32.1 ms
31.6 ms
Voltage Trace
Spike Peaks
inter-spike-intervals (ISI)
Vm (mV)
Time (ms)


CellId(population_name='S1nonbarrel_neurons', id=1)
24.2 ms
26.9 ms
Voltage Trace
Spike Peaks
inter-spike-intervals (ISI)
Vm (mV)
Time (ms)


CellId(population_name='S1nonbarrel_neurons', id=2)
36.3 ms
Vm (mV)
−10
−20
−30
−40
−50
−60
−70
−80
0
20
40
60
80
100
Time (ms)
Voltage Trace
Spike Peaks
inter-spike-intervals (ISI)

CHAPTER

THREE

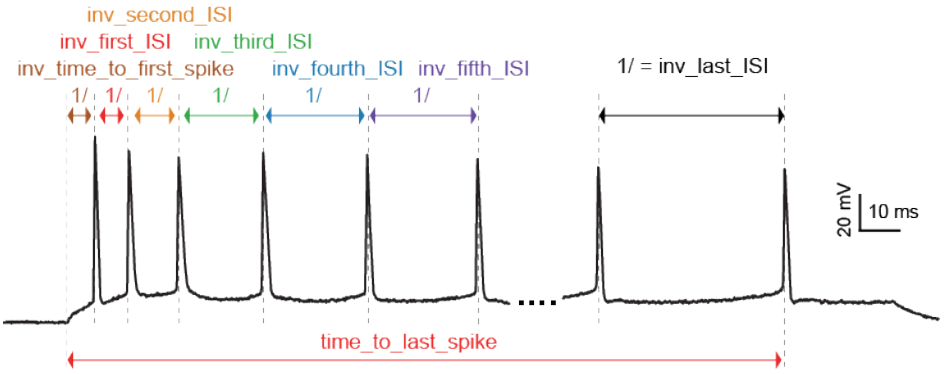
EFEATURE DESCRIPTIONS

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 = peaktime[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 = peaktime[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
```

Python efeature : ISI_values

The interspike intervals (i.e. time intervals) between adjacent peaks.

- **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)
```

Python efeature : 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/fith/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
```

Python efeature : inv_ISI_values

Computes all inverse spike interval values.

- **Required features**: peak_time (ms)
- **Units**: Hz
- **Pseudocode**:

```
all_isi_values_vec = numpy.diff(peak_time)
inv_isi_values = 1000.0 / all_isi_values_vec
```

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
```

Python efeature : spike_count

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

- **Required features**: LibV1:peak_indices
- **Units**: constant
- **Pseudocode**:

```
spike_count = len(peak_indices)
```

Note: “spike_count” is the new name for the feature “Spikecount”. “Spikecount”, while still available, will be removed in the future.

Python efeature : spike_count_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))
spike_count_stimint = len(peaktimes_stimint)
```

Note: “spike_count_stimint” is the new name for the feature “Spikecount_stimint”. “Spikecount_stimint”, while still available, will be removed in the future.

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 Python efeature: ISIs 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
```

Python efeature : 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 Python efeature: ISIs 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
```

Python efeature : 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
```

Python efeature : 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 Python efeature: ISIs feature for more details.

- **Required features**: ISIs
- **Units**: constant
- **Pseudocode**:

```
ISI_mean = numpy.mean(ISI_values)
ISI_CV = np.std(isi_values, ddof=1) / ISI_mean
```

Python efeature : irregularity_index

Mean of the absolute difference of all ISIs, except the first one (see Python efeature: ISIs 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)
```

Python efeature : 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]
        )
    )
```

Python efeature : burst_number

The number of bursts

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

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

Python efeature : 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)
```

Python efeature : 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)
]
```

Python efeature : 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 signle 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)
```

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. Defalt 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. Defalt 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. Defalt 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.

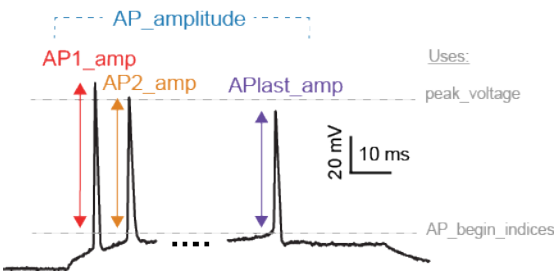
The first spike is ignored by default. This can be changed by setting ignore_first_ISI to 0.

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

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

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

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_height

Same as peak_voltage: The voltages at the maxima of the peaks

- **Required features**: LibV1:peak_voltage
- **Units**: mV
- **Pseudocode**:

```
AP_height = peak_voltage
```

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**:

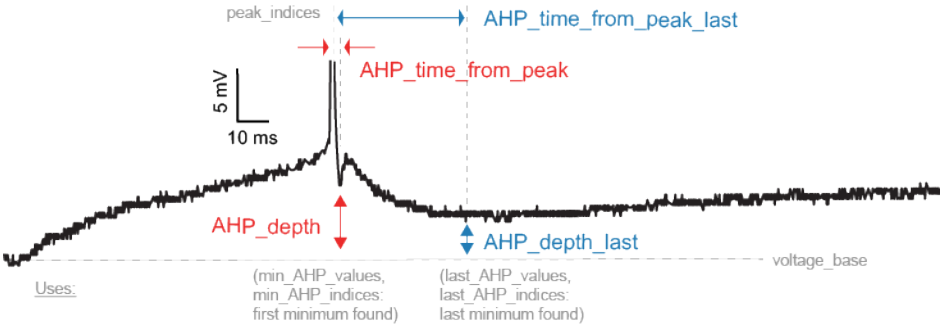
```
max_amp_difference = numpy.max(peak_voltage[:-1] - peak_voltage[1:])
```

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]))
```

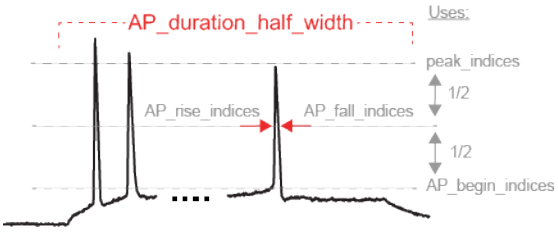
LibV5 : min_between_peaks_values

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_time[i] = t[numpy.where(v[onset_index:min_AHP_indices[i+1]] <␣
↪threshold)[0]]
    AP_width[i] = t(offset_time[i]) - t(onset_time[i])
```

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_
```

(continues on next page)

(continued from previous page)

```
↪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]
```

(continues on next page)

(continued from previous page)

```
    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, dVdt 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_idxs = AP_begin_indices + AP_phseslope_range
range_min_idxs = AP_begin_indices - AP_phseslope_range
AP_phaseslope = (dvdt[range_max_idxs] - dvdt[range_min_idxs]) / (v[range_max_idxs] -
↪ v[range_min_idxs])
```

Python efeature : phaseslope_max

Computes the maximum of the phase slope. Attention, this feature is sensitive to interpolation timestep.

- **Required features**: time, voltage
- **Units**: V/s
- **Pseudocode**:

```
phaseslope = numpy.diff(voltage) / numpy.diff(time)
phaseslope_max = numpy.array([numpy.max(phaseslope)])
```

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]])
```

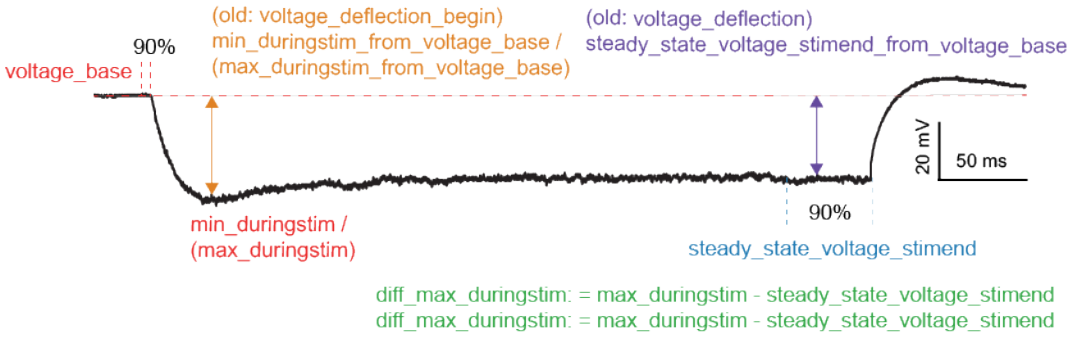
Python efeature : initburst_sahp_vb

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]])
```

3.1.3 Subthreshold 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.argwhere(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 follwowing 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

# 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]

# 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_idxs = numpy.append(
    -1, numpy.argwhere(dvdt > -min_derivative).flatten()
)
length_idx = numpy.argwhere(
    numpy.diff(pass_threshold_idxs) > decay_start_min_length
)[0][0]
i_start = pass_threshold_idxs[length_idx] + 1

# find flat (end of decay)
flat_idxs = numpy.argwhere(dvdt[i_start:] > -min_derivative).flatten()
# for loop is not optimised
```

(continues on next page)

(continued from previous page)

```
# but we expect the 1st few values to be the ones we are looking for
for i in flat_idxs:
    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
    )
    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)
    )
```

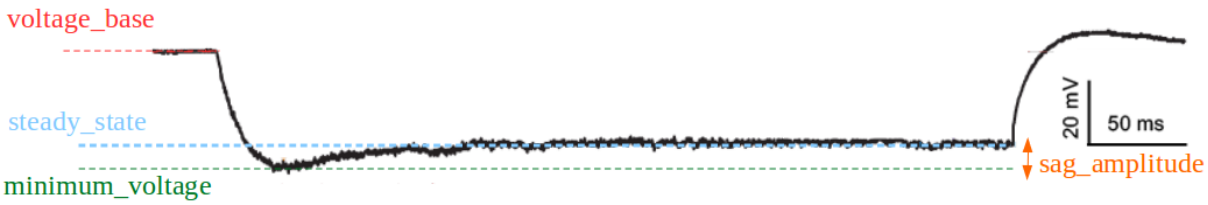
LibV5 : sag_time_constant

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(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 realtive 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 : 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 : 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, spike_count, 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
```

CHAPTER

FOUR

PYTHON API

4.1 Submodules

api	eFEL Python API functions.
io	
pyfeatures.cppfeature_access	Module containing access functions to C++ features for Python features.
pyfeatures.isi	Features that are depending on the inter-spike intervals.
pyfeatures.multitrace	Contains the features that are computed using multiple traces.
pyfeatures.pyfeatures	
pyfeatures.validation	Contains scientific validation methods on input signals.
units	Module to get units of efeatures.
settings	efel Settings class

4.1.1 efel.api

eFEL Python API functions.

This module provides the user-facing Python API of eFEL. The convenience functions defined here call the underlying 'cppcore' library to hide the lower level API from the user.

Functions

FeatureNameExists(feature_name) **param feature_name**	
feature_name_exists(feature_name)	Returns True if the feature name exists in eFEL, False otherwise.
getDependencyFileLocation() **rtype** str	
getDistance(trace, featureName, mean, std[, ...]) **rtype** float	
getFeatureNames() **rtype** list[str]	
getFeatureValues(traces, featureNames[, ...])	
getMeanFeatureValues(traces, featureNames[, ...])	
get_dependency_file_location()	Gets the location of the Dependency file.
get_distance(trace, feature_name, mean, std)	Calculate distance value for a list of traces.
get_feature_names()	Return a list with the name of all the available features
get_feature_values(traces, feature_names[, ...])	Calculate feature values for a list of traces.
get_mean_feature_values(traces, feature_names)	Convenience function that returns mean values from get_feature_values()
get_py_feature(feature_name)	Return values of the given feature name.
reset()	Resets the efel to its initial state
setDependencyFileLocation(location) **param location**	
setDerivativeThreshold(newDerivativeThreshold) **param newDerivativeThreshold**	
setDoubleSetting(setting_name, new_value) **param setting_name**	
setIntSetting(setting_name, new_value) **param setting_name**	
setStrSetting(setting_name, new_value) **param setting_name**	
setThreshold(newThreshold) **param newThreshold**	
set_dependency_file_location(location)	Sets the location of the Dependency file.
set_derivative_threshold(...)	Set the threshold for the derivative for detecting the spike onset.
set_double_setting(setting_name, new_value)	Set a certain double setting to a new value
set_int_setting(setting_name, new_value)	Set a certain integer setting to a new value
set_str_setting(setting_name, new_value)	Set a certain string setting to a new value
set_threshold(new_threshold)	Set the spike detection threshold in the eFEL, default 20.0

efel.api.**feature_name_exists**(*feature_name*)

Returns True if the feature name exists in eFEL, False otherwise.

Parameters
feature_name (str)

Return type
bool

efel.api.**get_dependency_file_location**()

Gets the location of the Dependency file.

Return type
str

Returns
Path to the location of a Dependency file.

efel.api.**get_distance**(*trace*, *feature_name*, *mean*, *std*, *trace_check=True*, *error_dist=250*)

Calculate distance value for a list of traces.

Parameters

- **trace** (dict) – Trace dict that represents one trace. The dict should have the following keys: 'T', 'V', 'stim_start', 'stim_end'
- **feature_name** (str) – 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** (bool) – 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

Return type
float

Returns
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.

efel.api.**get_feature_names**()

Return a list with the name of all the available features

Return type
list[str]

Returns
A list that contains all the feature names available in the eFEL. These names can be used in the feature_names argument of e.g. get_feature_values()

efel.api.**get_feature_values**(*traces*, *feature_names*, *parallel_map=None*, *return_list=True*, *raise_warnings=True*)

Calculate feature values for a list of traces.

This function is the core of 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[dict]) – Every trace dict represents one trace. The dict should have the following keys: 'T', 'V', 'stim_start', 'stim_end'
- **feature_names** (list[str]) – List with the names of the features to be calculated on all the traces.
- **parallel_map** (Optional[Callable]) – Map function to parallelise over the traces. Default is the serial map() function
- **return_list** (bool) – 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** (bool) – Raise warning when efel c++ returns an error

Return type
Union[list, Iterator]

Returns
For every input trace a feature value dict is returned (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.

efel.api.**get_mean_feature_values**(*traces*, *feature_names*, *raise_warnings=True*)

Convenience function that returns mean values from get_feature_values()

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

Parameters

- **traces** (list[dict]) – Every trace dict represents one trace. The dict should have the following keys: 'T', 'V', 'stim_start', 'stim_end'
- **feature_names** (list[str]) – List with the names of the features to be calculated on all the traces.
- **raise_warnings** (bool) – Raise warning when efel c++ returns an error

Return type
list[dict]

Returns
For every input trace a feature value dict is returned (in the same order). The dict contains the keys of 'feature_names', every key contains the mean of the array that is returned by get_feature_values() The value is None if an error occured during the calculation of the feature, or if the feature value array was empty.

efel.api.**get_py_feature**(*feature_name*)

Return values of the given feature name.

Parameters
feature_name (str)

Return type
ndarray | None

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.**set_dependency_file_location**(*location*)

Sets the location of the Dependency file.

eFEL uses 'Dependency' files to let the user define versions of features to use. The installation directory of eFEL contains a default 'DependencyV5.txt' file. Unless users want to change this file, it is not necessary to call this function. Modifying the Dependency file can be useful in debugging.

Parameters
location (str | Path) – Path to the location of a Dependency file.

Raises
FileNotFoundError – If the path to the dependency file doesn't exist.

Return type
None

efel.api.**set_derivative_threshold**(*new_derivative_threshold*)

Set the threshold for the derivative for detecting the spike onset.

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

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

Return type
None

efel.api.**set_double_setting**(*setting_name*, *new_value*)

Set a certain double setting to a new value

Parameters

- **setting_name** (str)
- **new_value** (float)

Return type
None

efel.api.**set_int_setting**(*setting_name*, *new_value*)

Set a certain integer setting to a new value

Parameters

- **setting_name** (str)
- **new_value** (int)

Return type
None

efel.api.**set_str_setting**(*setting_name*, *new_value*)

Set a certain string setting to a new value

Parameters

- **setting_name** (str)
- **new_value** (str)

Return type
None

efel.api.**set_threshold**(*new_threshold*)

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

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

Return type
None

4.1.2 efel.io

Functions

extract_stim_times_from_neo_data(blocks, ...)	Seeks for the stim_start and stim_end parameters inside the Neo data.
load_ascii_input(file_path[, delimiter])	Loads electrophysiology data from an ASCII file.
load_neo_file(file_name[, stim_start, stim_end])	Loads a data file using neo and converts it for eFEL readability.

efel.io.**extract_stim_times_from_neo_data**(*blocks*, *stim_start*, *stim_end*)

Seeks for the stim_start and stim_end parameters inside the Neo data.

Parameters

- **blocks** (*Neo object blocks*) – Description of what blocks represents.
- **stim_start** (*numerical value or None*) – Start time of the stimulation in milliseconds. If not available, None should be used.
- **stim_end** (*numerical value or None*) – End time of the stimulation in milliseconds. If not available, None should be used.

Returns

A tuple containing:

- stim_start (numerical value or None): Start time of the stimulation in milliseconds.
- stim_end (numerical value or None): End time of the stimulation in milliseconds.

Return type
tuple

Notes

- Epoch.name should be one of "stim", "stimulus", "stimulation", "current_injection".
- First Event.name should be "stim_start", "stimulus_start", "stimulation_start", "current_injection_start".
- Second Event.name should be one of "stim_end", "stimulus_end", "stimulation_end", "current_injection_end".

efel.io.**load_ascii_input**(*file_path*, *delimiter=' '*)

Loads electrophysiology data from an ASCII file.

Returns: A tuple containing two numpy arrays, one for time and one for voltage.

Parameters

- **file_path** (Path | str)
- **delimiter** (str)

Return type
tuple[ndarray, ndarray]

efel.io.**load_neo_file**(*file_name*, *stim_start=None*, *stim_end=None*, ***kwargs*)

Loads a data file using neo and converts it for eFEL readability.

Parameters

- **file_name** (*string*) – Path to the Dependency file location.
- **stim_start** (*numerical value, optional*) – Start time in ms. Optional if an Epoch or two Events are in the file.
- **stim_end** (*numerical value, optional*) – End time in ms. Optional if an Epoch or two Events are in the file.
- ****kwargs** – Additional arguments for the read() method of Neo IO class.

Returns

Segments containing traces, formatted as
[Segments_1, Segments_2, …, Segments_n], where each Segments_i is [Traces_1, Traces_2, …, Traces_n].

Return type
list of Segments

Notes

- Epoch.name should be "stim", "stimulus", "stimulation", "current_injection".
- First Event.name: "stim_start", "stimulus_start", "stimulation_start", "current_injection_start".
- Second Event.name: "stim_end", "stimulus_end", "stimulation_end", "current_injection_end".

4.1.3 efel.pyfeatures.cppfeature_access

Module containing access functions to C++ features for Python features.

Functions

get_cpp_feature(feature_name[, raise_warnings])	Return value of feature implemented in cpp.

efel.pyfeatures.cppfeature_access.**get_cpp_feature**(*feature_name*, *raise_warnings=False*)

Return value of feature implemented in cpp.

Parameters
feature_name (str)

Return type
ndarray | None

4.1.4 efel.pyfeatures.isi

Features that are depending on the inter-spike intervals.

Functions

ISI_CV()	Coefficient of variation of ISIs.
ISI_log_slope()	The slope of a linear fit to a loglog plot of the ISI values.
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.
ISI_semilog_slope()	The slope of a linear fit to a semilog plot of the ISI values.
ISI_values()	Get all ISIs, inter-spike intervals.
ISIs()	Get all ISIs, inter-spike intervals.
burst_ISI_indices()	Calculate burst ISI indices based on burst factor and ISI values.
burst_mean_freq()	Calculate the mean frequency of bursts.
initburst_sahp()	SlowAHP voltage after initial burst.
interburst_voltage()	The voltage average in between two bursts.
inv_ISI_values()	Calculate the inverse of ISI values.
inv_fifth_ISI()	Calculate the inverse of the fifth ISI.
inv_first_ISI()	Calculate the inverse of the first ISI.
inv_fourth_ISI()	Calculate the inverse of the fourth ISI.
inv_last_ISI()	Calculate the inverse of the last ISI.
inv_second_ISI()	Calculate the inverse of the second ISI.
inv_third_ISI()	Calculate the inverse of the third ISI.
irregularity_index()	Calculate the irregularity index of ISI values.
single_burst_ratio()	Calculates the single burst ratio.
strict_burst_number()	Calculate the strict burst number.

efel.pyfeatures.isi.**ISI_CV**()

Coefficient of variation of ISIs.

If the ignore_first_ISI flag is set, the first ISI will be ignored.

Return type
ndarray | None

efel.pyfeatures.isi.**ISI_log_slope**()

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

If the ignore_first_ISI flag is set, the first ISI will be ignored.

Return type
ndarray | None

efel.pyfeatures.isi.**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.

Uses the spike_skipf and max_spike_skip settings to determine how many ISIs to skip. .

Return type
ndarray | None

efel.pyfeatures.isi.**ISI_semilog_slope**()

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

If the ignore_first_ISI flag is set, the first ISI will be ignored.

Return type
ndarray | None

efel.pyfeatures.isi.**ISI_values**()

Get all ISIs, inter-spike intervals.

Return type
ndarray | None

efel.pyfeatures.isi.**ISIs**()

Get all ISIs, inter-spike intervals.

Return type
ndarray | None

efel.pyfeatures.isi.**burst_ISI_indices**()

Calculate burst ISI indices based on burst factor and ISI values.

Return type
ndarray | None

efel.pyfeatures.isi.**burst_mean_freq**()

Calculate the mean frequency of bursts.

Return type
ndarray | None

efel.pyfeatures.isi.**initburst_sahp**()

SlowAHP voltage after initial burst.

Return type
ndarray | None

efel.pyfeatures.isi.**interburst_voltage**()

The voltage average in between two bursts.

Return type
ndarray | None

efel.pyfeatures.isi.**inv_ISI_values**()

Calculate the inverse of ISI values.

Return type
ndarray | None

efel.pyfeatures.isi.**inv_fifth_ISI**()

Calculate the inverse of the fifth ISI.

Return type
ndarray | None

efel.pyfeatures.isi.**inv_first_ISI**()

Calculate the inverse of the first ISI.

Return type
ndarray | None

efel.pyfeatures.isi.**inv_fourth_ISI**()

Calculate the inverse of the fourth ISI.

Return type
ndarray | None

efel.pyfeatures.isi.**inv_last_ISI**()

Calculate the inverse of the last ISI.

Return type
ndarray | None

efel.pyfeatures.isi.**inv_second_ISI**()

Calculate the inverse of the second ISI.

Return type
ndarray | None

efel.pyfeatures.isi.**inv_third_ISI**()

Calculate the inverse of the third ISI.

Return type
ndarray | None

efel.pyfeatures.isi.**irregularity_index**()

Calculate the irregularity index of ISI values.

If the ignore_first_ISI flag is set, the first ISI will be ignored.

Return type
ndarray | None

efel.pyfeatures.isi.**single_burst_ratio**()

Calculates the single burst ratio.

The ratio is the length of the first ISI over the average of the rest. If the ignore_first_ISI flag is set, the first ISI will be ignored.

Return type
ndarray | None

efel.pyfeatures.isi.**strict_burst_number**()

Calculate the strict burst number.

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.

Return type
ndarray

4.1.5 efel.pyfeatures.multitrace

Contains the features that are computed using multiple traces.

Functions

bpap_attenuation(soma_trace, dendrite_trace)	Computes the attenuation of backpropagating action potential.

efel.pyfeatures.multitrace.**bpap_attenuation**(*soma_trace*, *dendrite_trace*)

Computes the attenuation of backpropagating action potential.

Backpropagating action potential is the action potential that is initiated in the soma and propagates to the dendrite. The attenuation is the ratio of the amplitude of the action potential in the soma and the dendrite. The attenuation is computed by first subtracting the resting potential from the voltage traces.

Parameters

- **soma_trace** (dict)
- **dendrite_trace** (dict)

Return type
float

4.1.6 efel.pyfeatures.pyfeatures

Functions

Spikecount() **rtype** ndarray	
Spikecount_stimint() **rtype** ndarray	
burst_number()	The number of bursts.
current()	Get current trace
depol_block()	Check for a depolarization block
depol_block_bool()	Wrapper around the depol_block feature.
impedance()	
initburst_sahp_ssse()	SlowAHP voltage from steady_state_voltage_stimend after initial burst
initburst_sahp_vb()	SlowAHP voltage from voltage base after initial burst
phaseslope_max()	Calculate the maximum phase slope.
spike_count()	Get spike count.
spike_count_stimint()	Get spike count within stimulus interval.
spikes_in_burst1_burst2_diff()	Calculate the diff between the spikes in 1st and 2nd bursts
spikes_in_burst1_burstlast_diff()	Calculate the diff between the spikes in 1st and last bursts
spikes_per_burst()	Calculate the number of spikes per burst
spikes_per_burst_diff()	Calculate the diff between the spikes in each burst and the next one
time()	Get time trace.
trace_check()	Returns np.array([0]) if there are no spikes outside stimulus boundaries.
voltage()	Get voltage trace.

efel.pyfeatures.pyfeatures.**burst_number**()

The number of bursts.

Return type
ndarray

efel.pyfeatures.pyfeatures.**current**()

Get current trace

efel.pyfeatures.pyfeatures.**depol_block**()

Check for a depolarization block

efel.pyfeatures.pyfeatures.**depol_block_bool**()

Wrapper around the depol_block feature. Returns [1] if depol_block is None, [0] otherwise.

efel.pyfeatures.pyfeatures.**initburst_sahp_ssse**()

SlowAHP voltage from steady_state_voltage_stimend after initial burst

efel.pyfeatures.pyfeatures.**initburst_sahp_vb**()

SlowAHP voltage from voltage base after initial burst

efel.pyfeatures.pyfeatures.**phaseslope_max**()

Calculate the maximum phase slope.

Return type
ndarray | None

efel.pyfeatures.pyfeatures.**spike_count**()

Get spike count.

Return type
ndarray

efel.pyfeatures.pyfeatures.**spike_count_stimint**()

Get spike count within stimulus interval.

Return type
ndarray

efel.pyfeatures.pyfeatures.**spikes_in_burst1_burst2_diff**()

Calculate the diff between the spikes in 1st and 2nd bursts

efel.pyfeatures.pyfeatures.**spikes_in_burst1_burstlast_diff**()

Calculate the diff between the spikes in 1st and last bursts

efel.pyfeatures.pyfeatures.**spikes_per_burst**()

Calculate the number of spikes per burst

efel.pyfeatures.pyfeatures.**spikes_per_burst_diff**()

Calculate the diff between the spikes in each burst and the next one

efel.pyfeatures.pyfeatures.**time**()

Get time trace.

Return type
ndarray | None

efel.pyfeatures.pyfeatures.**trace_check**()

Returns np.array([0]) if there are no spikes outside stimulus boundaries.

Returns None upon failure.

Return type
ndarray | None

efel.pyfeatures.pyfeatures.**voltage**()

Get voltage trace.

Return type
ndarray | None

4.1.7 efel.pyfeatures.validation

Contains scientific validation methods on input signals.

Functions

check_ais_initiation(soma_trace, ais_trace)	Checks the initiation of action potential in AIS with respect to soma.

efel.pyfeatures.validation.**check_ais_initiation**(*soma_trace*, *ais_trace*)

Checks the initiation of action potential in AIS with respect to soma.

Parameters

- **soma_trace** (dict)
- **ais_trace** (dict)

Return type
bool

4.1.8 efel.units

Module to get units of efeatures.

Functions

get_unit(feature_name)	Get the unit of a feature.

efel.units.**get_unit**(*feature_name*)

Get the unit of a feature.

Parameters
feature_name (str)

Return type
str

4.1.9 efel.settings

efel Settings class

Classes

Settings()	FEL settings class

class efel.settings.**Settings**

FEL settings class

CHAPTER

FIVE

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

5.1 5.6.6 - 2024-04

- Adding AP_height to documentation

5.2 5.6.0 - 2024-02

- Reduce 3 alternative implementations to get ISIs into 1.
- "all_ISI_values" is recommended, "ISI_values" and "ISIs" are deprecated.
- The features depending on "ISI_values" are moved to Python and now they depend on "all_ISI_values".
- BUGFIX: single_burst_ratio, irregularity_index, burst_mean_freq, interburst_voltage features were ignoring the first two ISIs when the ignore_first_ISI was set.
- Added new feature: inv_ISI_values that computes and returns all of the inverse isi values.

5.3 5.5.5 - 2024-01

- Type annotate api.py's functions.
- Deprecate camel case function names in api.py.
- Start using same requirements_docs.txt in readthedocs and tox.
- Enable autodoc and typehints in the API documentation.
- Fix docstring errors in the io module.
- Add changelog to the documentation.

5.4 [5.5.4] - 2024-01

- New feature: phaseslope_max

5.5 5.5.3 - 2024-01

- Add type stub for cppcore module to make Python recognise the C++ functions' arguments and return values.

5.6 5.5.0 - 2024-01

5.6.1 C++ changes

- AP_end_indices, AP_rise_time, AP_fall_time, AP_rise_rate, AP_fall_rate do not take into account peaks before stim_start anymore.
- New test and test data for spontaneous firing case. The data is provided by github user SzaBoglarka using cell https://modeldb.science/114047.

5.7 5.4.0 - 2024-01

5.7.1 C++ changes

- New C++ function *getFeatures* replaced *getVec*.
- *getFeatures* automatically handles failures & distinguishes empty results from failures.
- Centralized error handling in *getFeatures* shortens the code by removing repetitions.
- C++ features' access is restricted. Read-only references are marked *const*.
- Removed wildcard features from C++ API. Use of Python is encouraged for that purpose.

5.7.2 Python changes

- *bpap_attenuation* feature is added to the Python API.
- *Spikecount*, *Spikecount_stimint*, *burst_number*, *strict_burst_number* and *trace_check* features migrated to Python from C++.
- *check_ais_initiation* is added to the Python API.

CHAPTER

SIX

DEVELOPER'S GUIDE

Contents

6.1 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)

6.2 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/
```

6.3 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

6.4 Adding a new eFeature

Adding a new eFeature requires several steps.

6.4.1 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.

6.4.2 Creating a branch

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

```
git checkout -b your_efeaturename
```

6.4.3 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).

6.4.4 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
```

6.4.5 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
```

6.4.6 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
```

6.4.7 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/

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX