# eFEL Documentation

***Release 3.1.39***

**BBP, EPFL**

**Jun 03, 2020**

# Contents


**1 Installation** **3**
- 1.1 Requirements . . . 3
- 1.2 Installation using pip . . . 3
- 1.3 Installing the C++ standalone library . . . 4

**2 Examples** **5**
- 2.1 Python . . . 5
  - 2.1.1 Quick start . . . 5
  - 2.1.2 DEAP optimisation . . . 6
  - 2.1.3 Reading different file formats . . . 13

**3 eFeature descriptions** **15**
- 3.1 Implemented eFeatures (to be continued) . . . 15
  - 3.1.1 Spike event features . . . 15
  - 3.1.2 Spike shape features . . . 18
  - 3.1.3 Voltage features . . . 20
- 3.2 Requested eFeatures . . . 24

**4 Python API** **27**

**5 Developer's Guide** **31**
- 5.1 Requirements . . . 31
- 5.2 Forking and cloning the git repository . . . 32
- 5.3 Makefile . . . 32
- 5.4 Adding a new eFeature . . . 32
  - 5.4.1 Picking a name . . . 32
  - 5.4.2 Creating a branch . . . 32
  - 5.4.3 Implementation . . . 33
  - 5.4.4 Updating relevant files . . . 33
  - 5.4.5 Adding a test . . . 33
  - 5.4.6 Add documentation . . . 33
  - 5.4.7 Pull request . . . 34

**6 Indices and tables** **35**

**Python Module Index** **37**

Index 39

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

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

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

CHAPTER 1

# Installation

## 1.1 Requirements

- Python 2.7+ or Python 3.4+
- 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 git+git://github.com/BlueBrain/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 --user git+git://github.com/BlueBrain/eFEL
```

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 2

# Examples

## 2.1 Python

### 2.1.1 Quick start

First you need to import the module:

```
import efel
```

To get a list with all the available eFeature names:

```
efel.getFeatureNames()
```

The python function to extract eFeatures is getFeatureValues(...). Below is a short example on how to use this function.

The code and example trace are available here:

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

import efel
import numpy

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

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

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

(continues on next page)

(continued from previous page)

```
    # 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

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

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

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

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

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

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


if __name__ == '__main__':
    main()
```

The output of this example is:

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

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

The voltage before the start of the stimulis is measured by 'voltage_base'.

Results are in mV.

### 2.1.2 DEAP optimisation

**Contents**


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


## Introduction

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

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

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

The code of the example below can be downloaded from here

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

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

## Evaluation function

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

```
g_pas, e_pas
```

and the return values:

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

This translates into the following file (let's call it 'deap_efel_eval1.py'):

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

import efel

# pylint: disable=W0212


def evaluate(individual, target_voltage1=-80, target_voltage2=-60):
    """
```

(continues on next page)

(continued from previous page)

```
    Evaluate a neuron model with parameters e_pas and g_pas, extracts
    eFeatures from resulting traces and returns a tuple with
    abs(voltage_base-target_voltage1) and
    abs(steady_state_voltage-target_voltage2)
    """

    neuron.h.v_init = target_voltage1

    soma = neuron.h.Section()

    soma.insert('pas')

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

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

    stim_start = 500
    stim_end = 1000

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

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

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

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

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

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

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

### Setting up the algorithm

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

```
import random
import numpy

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

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

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

# Number of generations
NGEN = 300

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

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

We have two parameters with the following bounds:

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

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

As evolutionary algorithm we choose NSGA2:

```
SELECTOR = "NSGA2"
```

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

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

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

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

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

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

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

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

We first create the toolbox:

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

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

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

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

Now we can also register an individual:

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

And a population as list of individuals:

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

The function to evaluate we defined above. Assuming you saved that files as 'deap_efel_eval1.py', we can import it as a module, and register the function:

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

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

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

(continues on next page)

(continued from previous page)

```
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(
    "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__':
```

(continues on next page)

(continued from previous page)

```
    pop, logbook = algorithms.eaMuPlusLambda(
        pop,
        toolbox,
        MU,
        LAMBDA,
        CXPB,
        MUTPB,
        NGEN,
        stats,
        halloffame=None)
```

### 2.1.3 Reading different file formats

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

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

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

```
import efel

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

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

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

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

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

which we pass to eFEL as follows:

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

#### Stimulus information within the file

Some file formats can store information about the current injection stimulus. In this second example, the file contains an `Epoch` object named "stimulation", so we don't need to explicitly specify *stim_start* and *stim_end*:

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

# CHAPTER 3

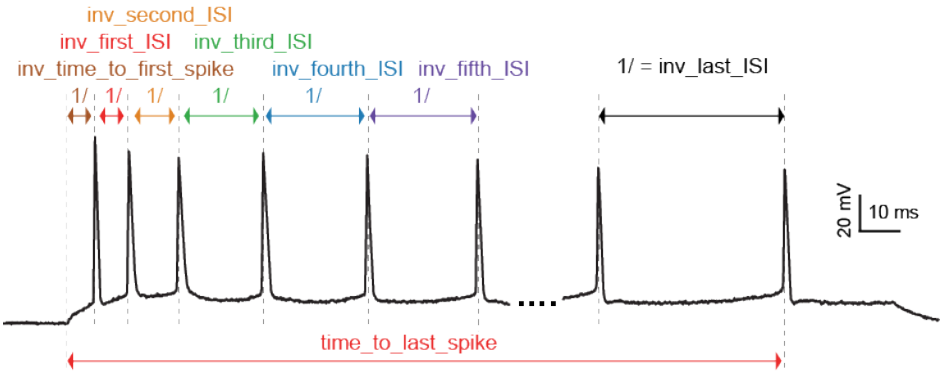
# eFeature descriptions

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




**LibV5 : inv_time_to_first_spike**

1.0 over time to first spike; 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 = 1.0 / time_to_first_spike[0]
else:
    inv_time_to_first_spike = 0
```

**LibV5 : inv_first_ISI, inv_second_ISI, inv_third_ISI, inv_fourth_ISI, inv_fith_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) > 1:
    inv_first_ISI = 1000.0 / all_isi_values_vec[0]
else:
    inv_first_ISI = 0

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

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

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

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

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

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

**LibV5 : time_to_last_spike**

time from stimulus start to last spike

- **Required features**: peak_time (ms), stimstart (ms)
- **Units**: ms
- **Pseudocode**:

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

**LibV1 : Spikecount**

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

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

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

**LibV5 : Spikecount_stimint**

number of spikes inside the stimulus interval

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

```
peaktimes_stimint = numpy.where((peak_time >= stim_start) & (peak_time <= stim_
↪end))
Spikecount_stimint = len(peaktimes_stimint)
```

**LibV5 : number_initial_spikes**

number of spikes at the beginning of the stimulus

- **Required features**: LibV1:peak_time
- **Required parameters**: initial_perc (default=0.1)
- **Units**: constant
- **Pseudocode**:

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

**LibV5 : ISI_semilog_slope**

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

- **Required features**: t, V, stim_start, stim_end, ISI_values
- **Units**: ms
- **Pseudocode**:

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

ISI_semilog_slope = slope
```

#### LibV5 : ISI_log_slope

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

- **Required features**: t, V, stim_start, stim_end, ISI_values
- **Units**: ms
- **Pseudocode**:

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

ISI_log_slope = slope
```

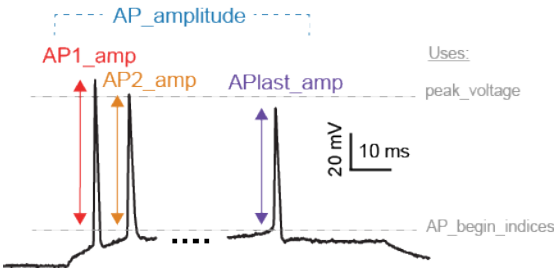
#### LibV5 : check_AISInitiation

Check initiation of AP in AIS

- **Required features**: t, V, stim_start, stim_end, AP_begin_time, AP_begin_time;location_AIS
- **Units**: ms
- **Pseudocode**:

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

### 3.1.2 Spike shape features

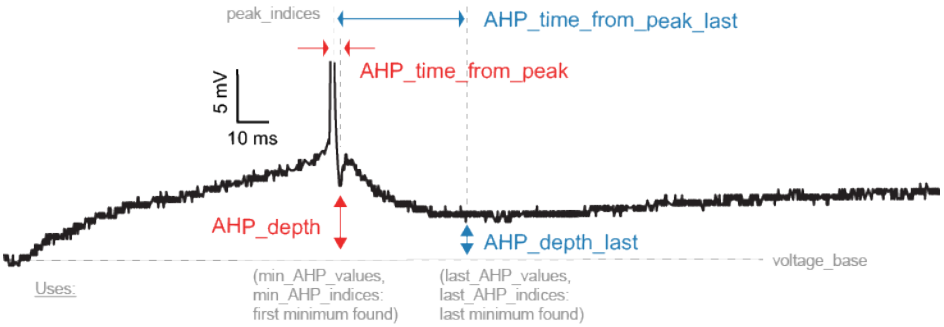



#### LibV1 : AP_Amplitude, AP1_amp, AP2_amp, APlast_amp

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




**LibV1 : AHP_depth_abs**

Absolute voltage values at the first after-hyperpolarization

- **Required features**: LibV5:min_AHP_values (mV)
- **Units**: mV

**LibV1 : AHP_depth_abs_slow**

Absolute voltage values at the first after-hyperpolarization starting 5 ms after the peak

- **Required features**: LibV1:peak_indices
- **Units**: mV

**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**: LibV1:voltage_base (mV), LibV5:min_AHP_values (mV)
- **Units**: mV
- **Pseudocode**:

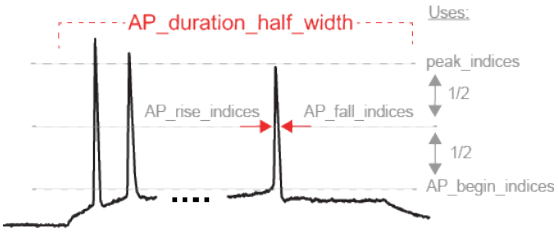
```
min_AHP_values = first_min_element(voltage, peak_indices)
AHP_depth = min_AHP_values[:] - voltage_base
```

**LibV5 : AHP_time_from_peak**

Time between AP peaks and first AHP depths

- **Required features**: LibV1:peak_indices, LibV5:min_AHP_values (mV)
- **Units**: mV
- **Pseudocode**:

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




**LibV2 : AP_duration_half_width**

Width of spike at half spike amplitude

- **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)
```

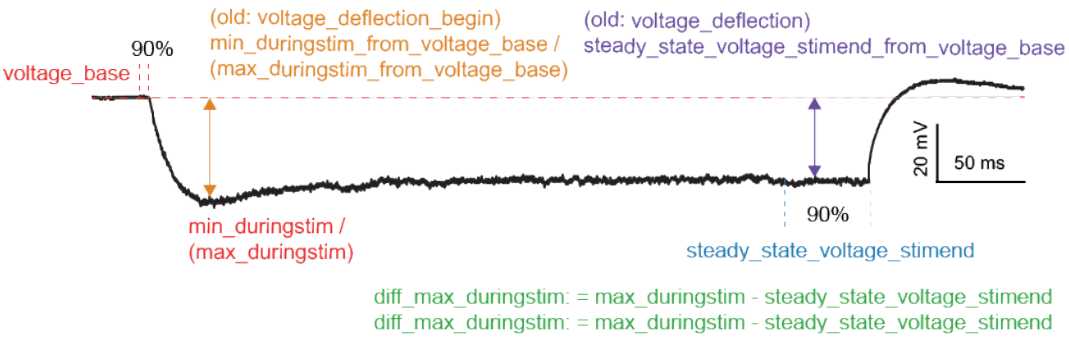
**LibV1 : AP_width**

Width of spike at threshold

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

```
min_AHP_indices.append(stim_start_index)
for i in range(len(min_AHP_indices)-1):
    onset_time[i] = t[numpy.where(v[min_AHP_indices[i]:min_AHP_indices[i+1]] >␣
↪threshold)[0]]
    offset_time[i] = t[numpy.where(v[min_AHP_indices[i]:min_AHP_indices[i+1]] <␣
↪threshold && t > onset_time)[0]]
    AP_width[i] = t(offset_time[i]) - t(onset_time[i])
```

### 3.1.3 Voltage features




#### LibV5 : steady_state_voltage_stimend

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

- **Required features**: t, V, stim_start, stim_end
- **Units**: mV
- **Pseudocode**:

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

### 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 : 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**:

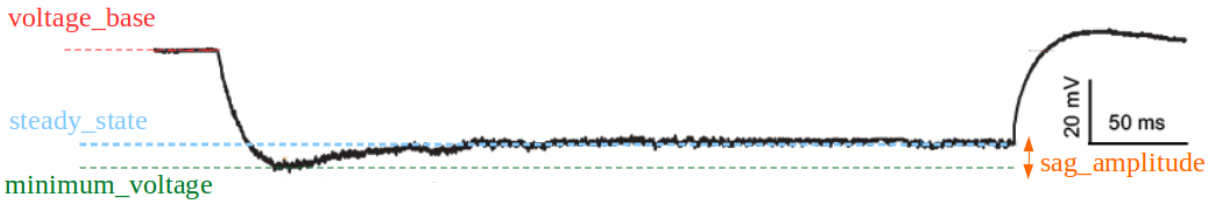
```
time_interval = t[numpy.where(t => decay_start_after_stim &
                    t < decay_end_after_stim)] - t[numpy.where(t == stim_end)]
voltage_interval = abs(voltages[numpy.where(t => decay_start_after_stim &
                                t < decay_end_after_stim)]
                       -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

(continues on next page)

(continued from previous page)

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

decay_time_constant_after_stim = -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**: mV/nA
- **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**: mV/nA
- **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

- **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
```

## 3.2 Requested eFeatures

**LibV1 : AHP_depth_last**

Relative voltage values at the last after-hyperpolarization

- **Required features**: LibV1: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**: mV
- **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** The minimum voltage during stimulus

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

```
min_duringstim = [numpy.min(voltage[numpy.where((t <= stim_end[0]) & (t >= stim_
↪start[0]))])]
```

**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 = min_duringstim - voltage_base
```

**LibV5 : max_duringstim** The minimum voltage during stimulus

- **Required features**:

- **Units**: mV
- **Pseudocode**:

```
min_duringstim = [numpy.max(voltage[numpy.where((t <= stim_end[0]) & (t >= stim_
↪start[0]))])]
```

**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 = max_duringstim - 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
```

# CHAPTER 4

# Python API

eFEL Python API functions.

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



efel.api.**FeatureNameExists**(*feature_name*)

Does a certain feature name exist ?

**Parameters** **feature_name** (*string*) – Name of the feature to check

**Returns** **FeatureNameExists** – True if feature_name exists, otherwise False

**Return type** bool

efel.api.**getDependencyFileLocation**()

Get the location of the Dependency file

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

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

**Return type** string

efel.api.**getDistance**(*trace*, *featureName*, *mean*, *std*, *trace_check=True*, *error_dist=250*)
Calculate distance value for a list of traces.

**Parameters**

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

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

**Return type** float

efel.api.**getFeatureNames**()
Return a list with the name of all the available features

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

**Return type** list of strings

efel.api.**getFeatureValues**(*traces*, *featureNames*, *parallel_map=None*, *return_list=True*, *raise_warnings=True*)
Calculate feature values for a list of traces.

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

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

**Parameters**

- **traces** (*list of trace dicts*) – Every trace dict represent one trace. The dict should have the following keys: 'T', 'V', 'stim_start', 'stim_end'
- **feature_names** (*list of string*) – List with the names of the features to be calculated on all the traces.
- **parallel_map** (*map function*) – Map function to parallelise over the traces. Default is the serial map() function
- **return_list** (*boolean*) – By default the function returns a list of dicts. This optional argument can disable this, so that the result of the parallel_map() is returned. Can be useful for performance reasons when an iterator is preferred.
- **raise_warnings** (*boolean*) – Raise warning when efel c++ returns an error

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

**Return type** list of dicts

efel.api.**getMeanFeatureValues**(*traces*, *featureNames*, *raise_warnings=True*)
Convenience function that returns mean values from getFeatureValues()

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

**Parameters**

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

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

**Return type** list of dicts

efel.api.**get_cpp_feature**(*featureName*, *raise_warnings=None*)
Return value of feature implemented in cpp

efel.api.**get_py_feature**(*featureName*)
Return python feature

efel.api.**reset**()
Resets the efel to its initial state

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

efel.api.**setDependencyFileLocation**(*location*)
Set the location of the Dependency file

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

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

efel.api.**setDerivativeThreshold**(*newDerivativeThreshold*)
Set the threshold for the derivate for detecting the spike onset

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

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

efel.api.**setDoubleSetting**(*setting_name*, *new_value*)
Set a certain double setting to a new value

efel.api.**setIntSetting**(*setting_name*, *new_value*)
Set a certain integer setting to a new value

efel.api.**setStrSetting**(*setting_name*, *new_value*)
Set a certain string setting to a new value

efel.api.**setThreshold**(*newThreshold*)
Set the spike detection threshold in the eFEL, default -20.0

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

CHAPTER 5

# Developer's Guide

**Contents**


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


## 5.1 Requirements

As a developer you will need some extra requirements

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

## 5.2 Forking and cloning the git repository

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

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

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

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

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

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

## 5.3 Makefile

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

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

## 5.4 Adding a new eFeature

Adding a new eFeature requires several steps.

### 5.4.1 Picking a name

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

### 5.4.2 Creating a branch

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

```
git checkout -b your_efeaturename
```

### 5.4.3 Implementation

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

### 5.4.4 Updating relevant files

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

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

You can confirm everything compiles correctly by executing:

```
make test
```

### 5.4.5 Adding a test

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

The Nose tests of the individual eFeatures are here .Just add your own test by defining a new function 'test_yourfeature()'.

Some test data is available here , but you can of course add your own traces.

The easiest way to run the tests is by executing:

```
make test
```

### 5.4.6 Add documentation

Add the documentation of the new eFeature to this file:

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

Please provide some pseudo-Python code for the eFeature.

The documentation can be built by:

```
make doc
```

It can be viewed by opening:

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

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

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

### 5.4.7 Pull request

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

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

Finally create a pull request:

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

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## e

efel.api, 27

# Index


## E

efel.api (*module*), 27

## F

FeatureNameExists() (*in module efel.api*), 27

## G

get_cpp_feature() (*in module efel.api*), 29
get_py_feature() (*in module efel.api*), 29
getDependencyFileLocation() (*in module efel.api*), 27
getDistance() (*in module efel.api*), 27
getFeatureNames() (*in module efel.api*), 28
getFeatureValues() (*in module efel.api*), 28
getMeanFeatureValues() (*in module efel.api*), 29

## R

reset() (*in module efel.api*), 29

## S

setDependencyFileLocation() (*in module efel.api*), 29
setDerivativeThreshold() (*in module efel.api*), 29
setDoubleSetting() (*in module efel.api*), 29
setIntSetting() (*in module efel.api*), 29
setStrSetting() (*in module efel.api*), 30
setThreshold() (*in module efel.api*), 30