Brian2modelfitting

Release 0.3

Jun 18, 2020

Contents

The package *brian2modelfitting* is a tool for parameter fitting of neuron models in the Brian 2 simulator.

Please contact us at brian-development@googlegroups.com (https://groups.google.com/forum/#!forum/brian-development) if you are interested in contributing.

Please report bugs at the github issue tracker or to briansupport@googlegroups.com (https://groups.google.com/forum/#!forum/briansupport).

CHAPTER 1

Model fitting

The `brian2modelfitting` toolbox offers allows the user to perform data driven optimization for custom neuronal models specified with Brian 2.

The toolbox allows the user to find the best fit of the parameters for recorded traces and spike trains. Just like Brian itself, the Model Fitting Toolbox is designed to be easy to use and save time through automatic parallelization of the simulations using code generation.

CHAPTER 2

Contents

2.1 Model Fitting

2.1.1 Introduction

The *brian2modelfitting* toolbox provides three optimization classes:

- *TraceFitter*
- *SpikeFitter*
- *OnlineTraceFitter*

The classes expect a model and data as an input and returns the best fit of parameters and the corresponding error. The toolbox can optimize over multiple traces (e.g. input currents) at the same time.

In following documentation we assume that `brian2modelfitting` has been imported like this:

```
from brian2modelfitting import *
```

Installation

To install Model Fitting alongside Brian2 you can use pip, by using a pip utility:

```
pip install brian2modelfitting
```

Testing Model Fitting

Version on master branch gets automatically tested with Travis services. To test the code yourself, you will need to have `pytest` installed and run the command:

```
pytest
```

2.1.2 How it works

Model fitting script requires three components:

- A **Fitter** of choice: object that will perform the optimization
- A **metric**: to compare results and decide which one is the best
- An **optimization** algorithm: to decide which parameter combinations to try

All of which need to be initialized for fitting application. Each optimization works with a following scheme:

```
opt = Optimizer()
metric = Metric()

fitter = Fitter(...)
result, error = fitter.fit(metric=metric, optimizer=opt, ...)
```

The proposed solution is developed using a modular approach, where both the optimization method and metric to be optimized can be easily swapped out by a custom implementation.

Fitter objects require 'model' defined as an `Equations` object or as a string, that has parameters that will be optimized specified as constants in the following way:

```
model = '''
...
g_na : siemens (constant)
g_kd : siemens (constant)
gl   : siemens (constant)
'''
```

Initialization of Fitter requires:

- `dt` - time step
- `input` - set of input traces (list or array)
- `output` - set of goal output (traces/spike trains) (list or array)
- `input_var` - name of the input trace variable (string)
- `output_var` - name of the output trace variable (string)
- `n_samples` - number of samples to draw in each round (limited by method)
- `reset`, and `threshold` in case of spiking neurons (can take refractory as well)

Additionally, upon call of *fit()*, object requires:

- `n_rounds` - number of rounds to optimize over
- parameters with ranges to be optimized over

. . . as well as an `optimizer` and a `metric`

Each free parameter of the model that shall be fitted is defined by two values:

```
param_name = [min, max]
```

Ready to use elements

Alongside three optimization classes:

- *TraceFitter*
- *SpikeFitter*
- *OnlineTraceFitter*

We also provide ready optimizers:

- *NevergradOptimizer*
- *SkoptOptimizer*

and metrics:

- *MSEMetric* (for *TraceFitter*)
- *GammaFactor* (for *SpikeFitter*)

Example of *TraceFitter* with all of the necessary arguments:

```
fitter = TraceFitter(model=model,
                     input=inp_traces,
                     output=out_traces,
                     input_var='I',
                     output_var='v',
                     dt=0.1*ms,
                     n_samples=5)

result, error = fitter.fit(optimizer=optimizer,
                           metric=metric,
                           n_rounds=1,
                           gl=[1e-8*siemens*cm**-2 * area, 1e-3*siemens*cm**-2 *␣
↪area],)
```

Remarks

- After performing first fitting, user can continue the optimization with another *fit()* run.
- Number of samples can not be changed between rounds or *fit()* calls, due to parallelization of the simulations.

Warning: User is not allowed to change the optimizer or metric between *fit()* calls.

- When using the *TraceFitter*, users can use a standard curve fitting algorithm for refinement by calling *refine*.

2.1.3 Tutorial: TraceFitter

In following documentation we will explain how to get started with using *TraceFitter*. Here we will optimize conductances for a Hodgkin-Huxley cell model.

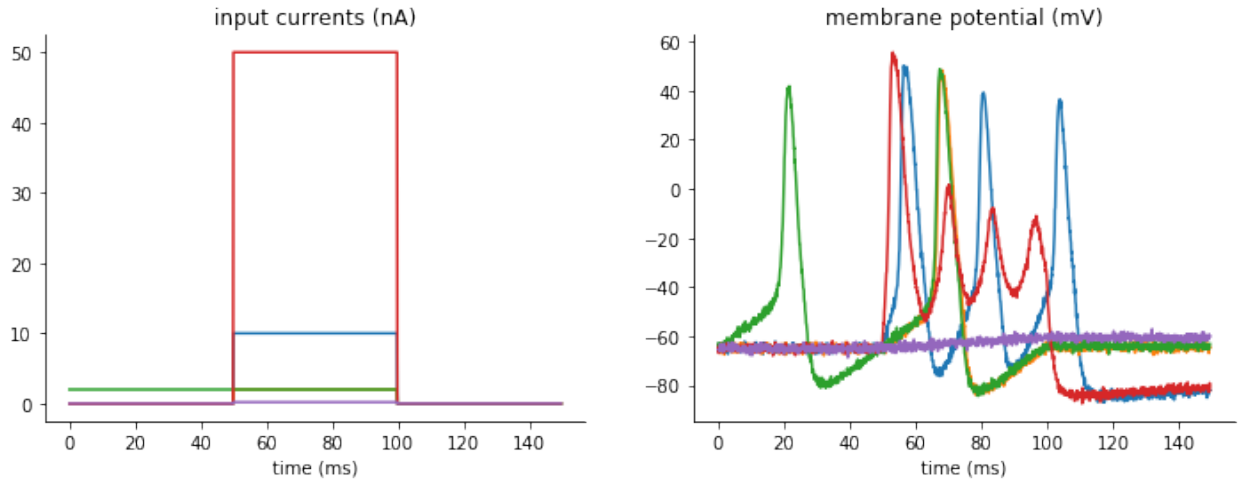
We start by importing `brian2` and `brian2modelfitting`:

```
from brian2 import *
from brian2modelfitting import *
```

Problem description

We have five step input currents of different amplitude and five "data samples" recorded from the model with goal parameters. The goal of this exercise is to optimize the conductances of the model `gl`, `g_na`, `g_kd`, for which we know the expected ranges.

Visualization of input currents and corresponding output traces which we will try to fit:



We can load these currents and "recorded" membrane potentials with the pandas library

```
import pandas as pd
inp_trace = pd.read_csv('input_traces_hh.csv', index_col=0).to_numpy()
out_trace = pd.read_csv('output_traces_hh.csv', index_col=0).to_numpy()
```

Note: You can download the CSV files used above here: `input_traces_hh.csv`, `output_traces_hh.csv`

Procedure

Model definition

We have to specify all of the constants for the model

```
area = 20000*umetre**2
Cm=1*ufarad*cm**-2 * area
El=-65*mV
EK=-90*mV
ENa=50*mV
VT=-63*mV
```

Then, we have to define our model:

```
model = '''
dv/dt = (gl*(El-v) - g_na*(m*m*m)*h*(v-ENa) - g_kd*(n*n*n*n)*(v-EK) + I)/Cm : volt
dm/dt = 0.32*(mV**-1)*(13.*mV-v+VT)/
  (exp((13.*mV-v+VT)/(4.*mV))-1.)/ms*(1-m)-0.28*(mV**-1)*(v-VT-40.*mV)/
  (exp((v-VT-40.*mV)/(5.*mV))-1.)/ms*m : 1
dn/dt = 0.032*(mV**-1)*(15.*mV-v+VT)/
```

(continues on next page)

(continued from previous page)

```
  (exp((15.*mV-v+VT)/(5.*mV))-1.)/ms*(1.-n)-.5*exp((10.*mV-v+VT)/(40.*mV))/ms*n : 1
dh/dt = 0.128*exp((17.*mV-v+VT)/(18.*mV))/ms*(1.-h)-4./(1+exp((40.*mV-v+VT)/(5.*mV)))/
↪ms*h : 1
g_na : siemens (constant)
g_kd : siemens (constant)
gl   : siemens (constant)
'''
```

Note: You have to identify the parameters you want to optimize by adding them as constant variables to the equation.

Optimizer and metric

Once we know our model and parameters, it's time to pick an optimizing algorithm and a metric that will be used as a measure.

For simplicity we will use the default method provided by the *NevergradOptimizer*, i.e. "Differential Evolution", and the *MSEMetric*, calculating the mean squared error between simulated and data traces:

```
opt = NevergradOptimizer()
metric = MSEMetric()
```

Fitter Initiation

Since we are going to optimize over traces produced by the model, we need to initiate the fitter *TraceFitter*: The minimum set of input parameters for the fitter, includes the `model` definition, `input` and `output` variable names and traces, time step `dt`, number of samples we want to draw in each optimization round.

```
fitter = TraceFitter(model=model,
                     input_var='I',
                     output_var='v',
                     input=inp_trace * amp,
                     output=out_trace*mV,
                     dt=0.01*ms,
                     n_samples=100,
                     method='exponential_euler',
                     param_init={'v': -65*mV})
```

Additionally, in this example, we pick the integration method to be `'exponential_euler'`, and we specify the initial value of the state variable `v`, by using the option: `param_init={'v': -65*mV}`.

Fit

We are now ready to perform the optimization, by calling the *fit* method. We need to pass the `optimizer`, `metric` and pick a number of rounds(`n_rounds`).

Note: Here you have to also pass the ranges for each of the parameters that was defined as a constant!

```
res, error = fitter.fit(n_rounds=10,
                        optimizer=opt,
                        metric=metric,
                        gl=[2*psiemens, 200*nsiemens],
                        g_na=[200*nsiemens, 0.4*msiemens],
                        g_kd=[200*nsiemens, 200*usiemens])
```

Output:

- `res`: dictionary with best fit values from this optimization
- `error`: corresponding error

The default output during the optimization run will tell us the best parameters in each round of optimization and the corresponding error:

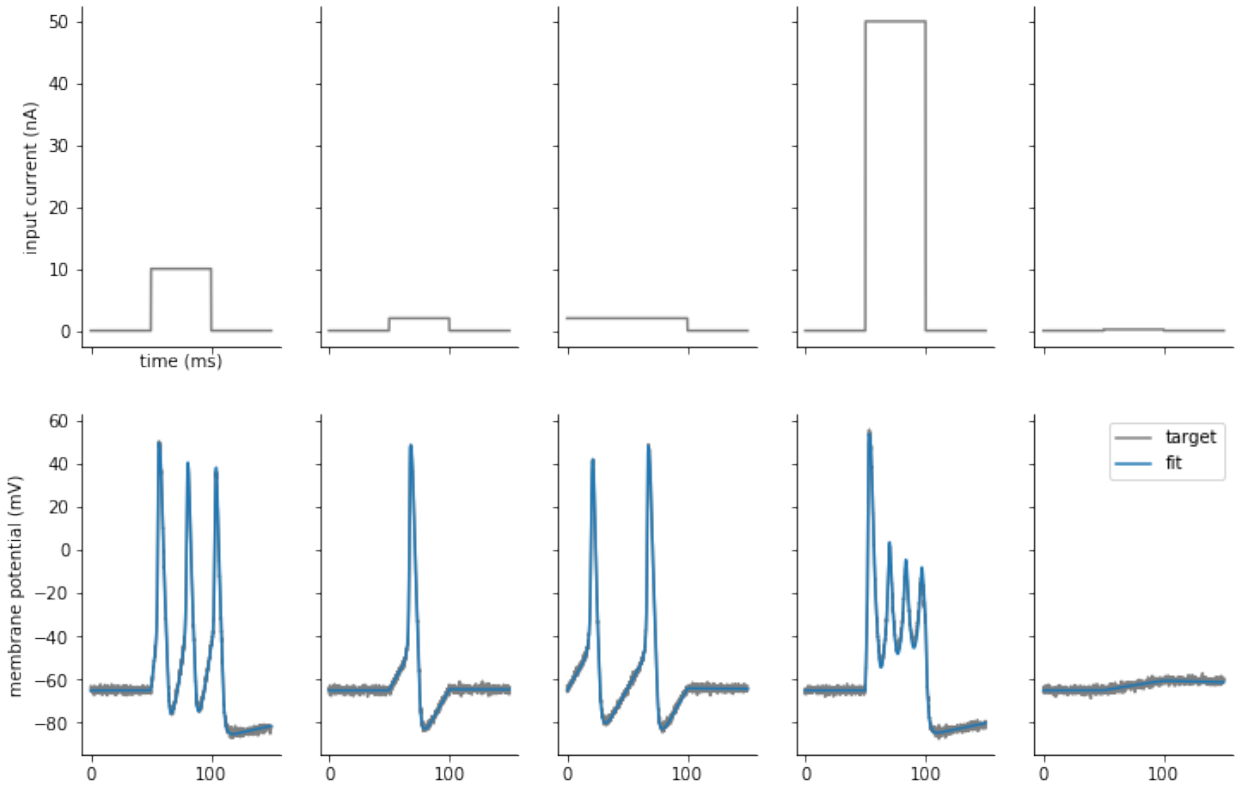
```
Round 0: fit [9.850944960633812e-05, 5.136956717618642e-05, 1.132001753695881e-07]␣
↪with error: 0.00023112503428419085
Round 1: fit [2.5885625978001192e-05, 5.994175009416155e-05, 1.132001753695881e-07]␣
↪with error: 0.0001351283127819249
Round 2: fit [2.358033085911261e-05, 5.286319601683492e-05, 7.255743458079185e-08]␣
↪with error: 8.600916130059129e-05
Round 3: fit [2.013515980650059e-05, 4.588859295919631e-05, 7.3254174819061e-08]␣
↪with error: 5.704891495098806e-05
Round 4: fit [9.666300621928093e-06, 3.471303670631636e-05, 2.6927249265934296e-08]␣
↪with error: 3.237910401003197e-05
Round 5: fit [8.037164838105382e-06, 2.155149445338687e-05, 1.9305129338706338e-08]␣
↪with error: 1.080794896277778e-05
Round 6: fit [7.161113899555702e-06, 2.2680883630214104e-05, 2.369859751788268e-08]␣
↪with error: 4.527456021770018e-06
Round 7: fit [7.471475084450997e-06, 2.3920164839406964e-05, 1.7956856689140395e-08]␣
↪with error: 4.4765688852930405e-06
Round 8: fit [6.511156620775884e-06, 2.209792671051356e-05, 1.368667359118384e-08]␣
↪with error: 1.8105782339584402e-06
Round 9: fit [6.511156620775884e-06, 2.209792671051356e-05, 1.368667359118384e-08]␣
↪with error: 1.8105782339584402e-06
```

Generating traces

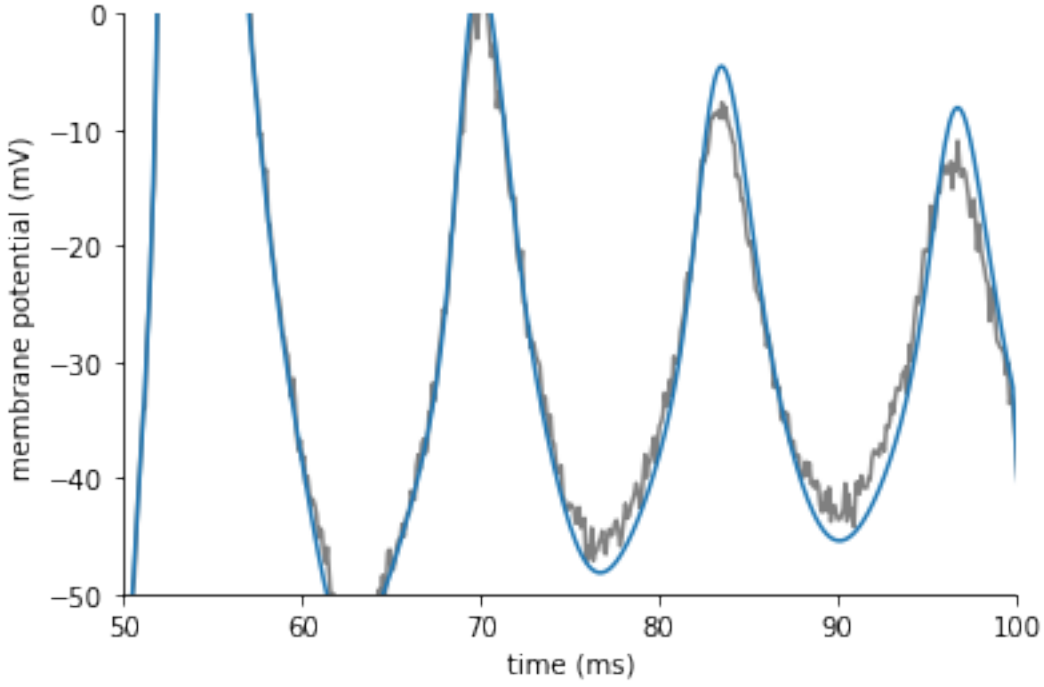
To generate the traces that correspond to the new best fit parameters of the model, you can use the *generate_traces* method.

```
traces = fitter.generate_traces()
```

The following plot shows the fit traces in comparison to our target data:



The fit looks good in general, but if we zoom in on the fourth column we see that the fit is still not perfect:



We can improve the fit by using a classic, sequential curve fitting algorithm.

Refining fits

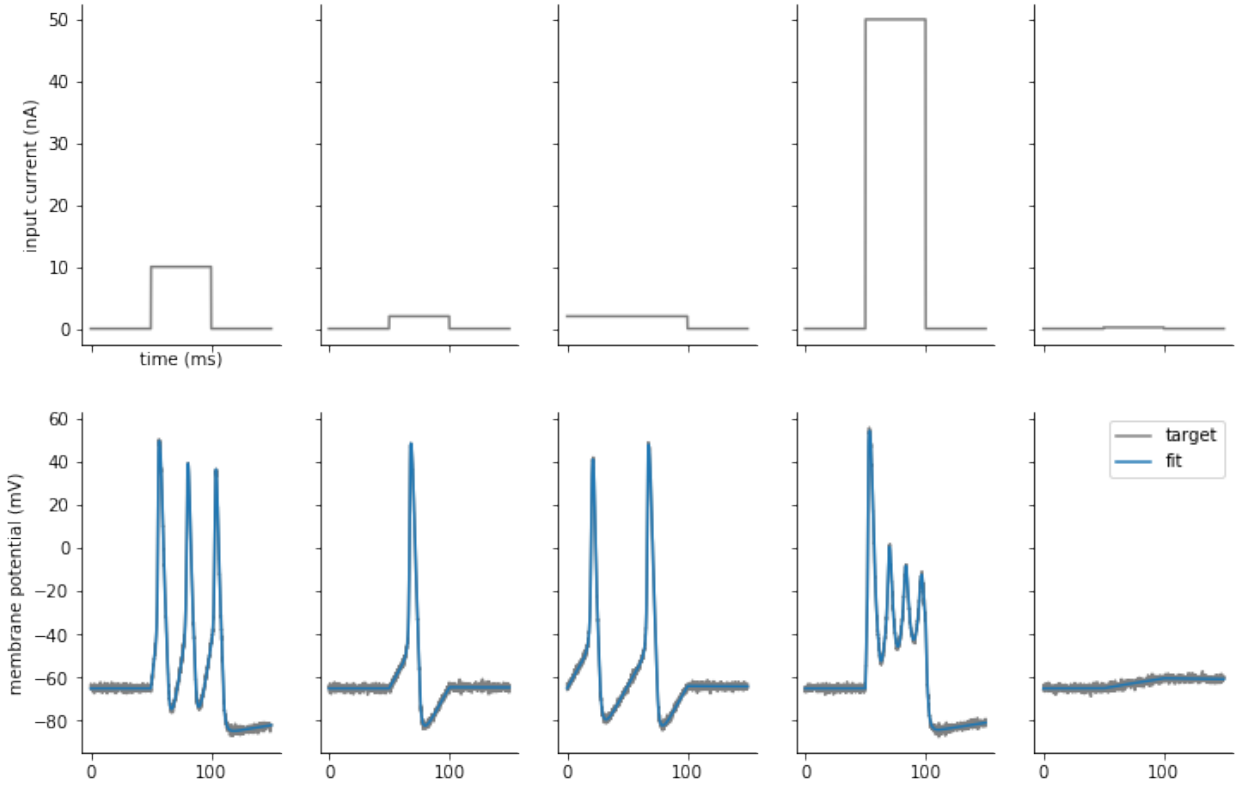
When using *TraceFitter*, you can further refine the fit by applying a standard least squares fitting algorithm (e.g. Levenberg–Marquardt), by calling *refine*. By default, this will start from the previously found best parameters:

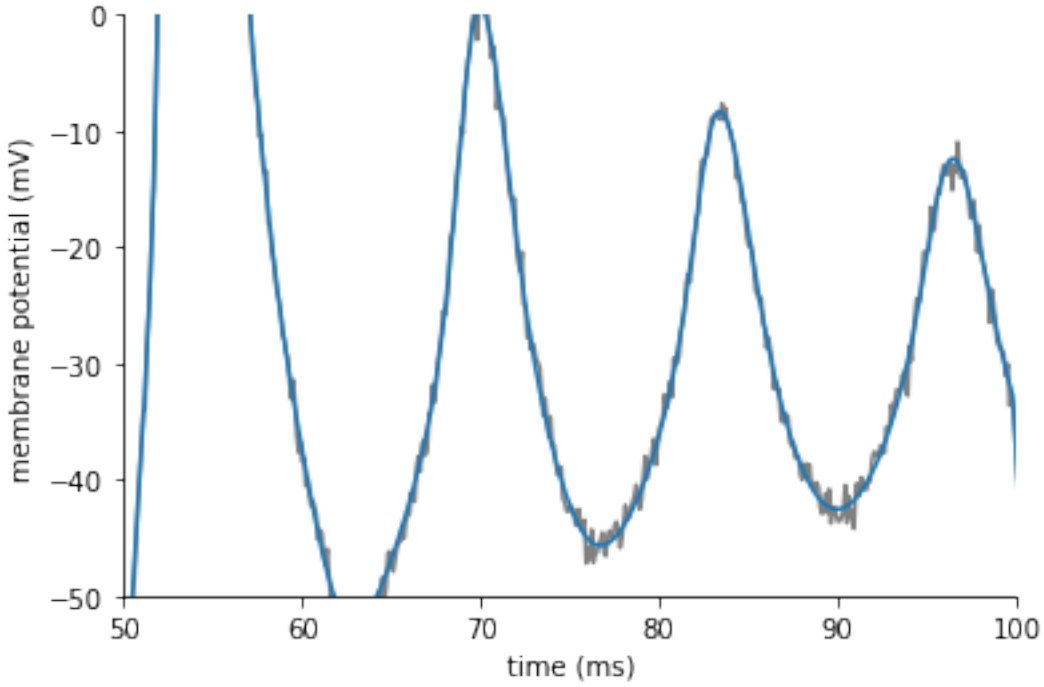
```
refined_params, result_info = fitter.refine()
```

We can now generate traces with the refined parameters:

```
traces = fitter.generate_traces(params=refined_params)
```

Plotting the results, we see that the fits have improved and now closely match the target data:





2.2 Optimizer

Optimizer class is responsible for maximizing a fitness function. Our approach uses gradient free global optimization methods (evolutionary algorithms, genetic algorithms, Bayesian optimization). We provided access to two libraries.

2.2.1 Nevergrad

Offers an extensive collection of algorithms that do not require gradient computation. `NevergradOptimizer` can be specified in the following way:

```
opt = NevergradOptimizer(method='PSO')
```

where method input is a string with specific optimization algorithm.

Available methods include:

- Differential evolution. `['DE']`
- Covariance matrix adaptation.`['CMA']`
- Particle swarm optimization.`['PSO']`
- Sequential quadratic programming.`['SQP']`

Nevergrad is still poorly documented, to check all the available methods use the following code:

```
from nevergrad.optimization import registry
print(sorted(registry.keys()))
```

2.2.2 Scikit-Optimize (skopt)

Skopt implements several methods for sequential model-based ("blackbox") optimization and focuses on bayesian methods. Algorithms are based on scikit-learn minimize function.

Available Methods:

- Gaussian process-based minimization algorithms ['GP']
- Sequential optimization using gradient boosted trees ['GBRT']
- Sequential optimisation using decision trees ['ET']
- Random forest regressor ['RF']

User can also provide a custom made sklearn regressor. *SkoptOptimizer* can be specified in the following way:

Parameters:

- method = ["GP", "RF", "ET", "GBRT" or sklearn regressor, default="GP"]
- n_initial_points [int, default=10]
- acq_func
- acq_optimizer
- random_state

For more detail check Optimizer documentation. https://scikit-optimize.github.io/#skopt.Optimizer

```
opt = SkoptOptimizer(method='GP', acq_func='LCB')
```

2.2.3 Custom Optimizer

To use a different back-end optimization library, user can provide a custom class that inherits from provided abstract class *Optimizer*

User can plug in different optimization tool, as long as it follows an `ask() / tell` interface. The abstract class *Optimizer* is prepared for different back-end libraries. All of the optimizer specific arguments have to be provided upon optimizers initialization.

The `ask() / tell` interface is used as follows:

```
parameters = optimizer.ask()

errors = simulator.run(parameters)

optimizer.tell(parameters, errors)
results = optimizer.recommend()
```

2.3 Metric

A *Metric* specifies the fitness function measuring the performance of the simulation. This function gets applied on each simulated trace. A few metrics are already implemented and included in the toolbox, but the user can also provide their own metric.

2.3.1 Mean Square Error

`MSEMetric` is provided for use with `TraceFitter`. It calculates the mean squared difference between the data and the simulated trace according to the well known formula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

It can be initialized in the following way:

```
metric = MSEMetric()
```

Additionally, `MSEMetric` accepts an optional input argument start time `t_start` (as a `Quantity`). The start time allows the user to ignore an initial period that will not be included in the error calculation.

```
metric = MSEMetric(t_start=5*ms)
```

Alternatively, the user can specify a weight vector emphasizing/de-emphasizing certain parts of the trace. For example, to ignore the first 5ms and to weigh the error (in the sense of the squared error) between 10 and 15ms twice as high as the rest:

```
# total trace length = 50ms
weights = np.ones(int(50*ms/dt))
weights[:int(5*ms/dt)] = 0
weights[int(10*ms/dt):int(15*ms/dt)] = 2
metric = MSEMetric(t_weights=weights)
```

Note that the `t_weights` argument cannot be combined with `t_start`.

In `OnlineTraceFitter`, the mean square error gets calculated in online manner, with no need of specifying a metric object.

2.3.2 GammaFactor

`GammaFactor` is provided for use with `SpikeFitter` and measures the coincidence between spike times in the simulated and the target trace. It is calculcated according to:

$$\Gamma = \left(\frac{2}{1 - 2\Delta r_{exp}}\right)\left(\frac{N_{coinc} - 2\delta N_{exp} r_{exp}}{N_{exp} + N_{model}}\right)$$

N_{coinc} - number of coincidences

N_{exp} and N_{model}- number of spikes in experimental and model spike trains

r_{exp} - average firing rate in experimental train

$2\Delta N_{exp} r_{exp}$ - expected number of coincidences with a Poission process

For more details on the gamma factor, see:

- Jolivet et al. 2008, "A benchmark test for a quantitative assessment of simple neuron models", J. Neurosci. Methods.
- Clopath et al. 2007, "Predicting neuronal activity with simple models of the threshold type: adaptive exponential integrate-and-fire model with two compartments.", Neurocomp

The coincidence factor Γ is 1 if the two spike trains match exactly and lower otherwise. It is 0 if the number of coincidences matches the number expected from two homogeneous Poisson processes of the same rate. To turn the coincidence factor into an error term (that is lower for better matches), two options are offered. With the `rate_correction` option (used by default), the error term used is $2\frac{|r_{\mathrm{data}}-r_{\mathrm{model}}|}{r_{\mathrm{data}}} - \Gamma$, with r_{data} and r_{model} being the firing rates in the data/model. This is useful because the coincidence factor Γ on its own can give high values (low errors) if the model generates many more spikes than were observed in the data; this is penalized by the above term. If `rate_correction` is set to `False`, $1 - \Gamma$ is used as the error.

Upon initialization the user has to specify the Δ value, defining the maximal tolerance for spikes to be considered coincident:

```
metric = GammaFactor(delta=2*ms)
```

Warning: The `delta` parameter has to be smaller than the smallest inter-spike interval in the spike trains.

2.3.3 FeatureMetric

FeatureMetric is provided for use with *TraceFitter*. This metric allows the user to optimize the match of certain features between the simulated and the target trace. The features get calculated by Electrophys Feature Extract Library (eFEL) library, for which the documentation is available under following link: https://efel.readthedocs.io

To get a list of all the available eFEL features, you can run the following code:

```
import efel
efel.api.getFeatureNames()
```

Note: Currently, only features that are described by a single value are supported (e.g. the time of the first spike can be used, but not the times of all spikes).

To use the *FeatureMetric*, you have to provide the following input parameters:

- `stim_times` - a list of times indicating start and end of the stimulus for each of input traces. This information is used by several features, e.g. the `voltage_base` feature will consider the average membrane potential during the last 10% of time before the stimulus (see the eFel documentation for details).
- `feat_list` - list of strings with names of features to be used
- `combine` - function to be used to compare features between output and simulated traces (uses the absolute difference between the values by default).

Example code usage:

```
stim_times = [(50*ms, 100*ms), (50*ms, 100*ms), (50*ms, 100*ms), (50, 100*ms)]
feat_list = ['voltage_base', 'time_to_first_spike', 'Spikecount']
metric = FeatureMetric(traces_times, feat_list, combine=None)
```

Note: If times of stimulation are the same for all of the traces, then you can specify a single interval instead: `traces_times = [(50*ms, 100*ms)]`.

2.3.4 Custom Metric

Users are not limited to the metrics provided in the toolbox. If needed, they can provide their own metric based on one of the abstract classes *TraceMetric* and *SpikeMetric*.

A new metric will need to specify the following functions:

- ***get_features()*** calculates features / errors for each of the simulations. The representation of the model results and the target data depend on whether traces or spikes are fitted, see below.
- ***get_errors()*** weights features/multiple errors into one final error per each set of parameters and inputs. The features are received as a 2-dimensional `ndarray` of shape `(n_samples, n_traces)` The output has to be an array of length `n_samples`, i.e. one value for each parameter set.
- ***calc()*** performs the error calculation across simulation for all parameters of each round. Already implemented in the abstract class and therefore does not need to be reimplemented necessarily.

TraceMetric

To create a new metric for *TraceFitter*, you have to inherit from *TraceMetric* and overwrite the *get_features* and/or *get_errors* method. The model traces for the *get_features* function are provided as a 3-dimensional `ndarray` of shape `(n_samples, n_traces, time steps)`, where `n_samples` are the number of different parameter sets that have been evaluated, and `n_traces` the number of different stimuli that have been evaluated for each parameter set. The output of the function has to take the shape of `(n_samples, n_traces)`. This array is the input to the *get_errors* method (see above).

```
class NewTraceMetric(TraceMetric):
  def get_features(self, model_traces, data_traces, dt):
    ...

  def get_errors(self, features):
    ...
```

SpikeMetric

To create a new metric for *SpikeFitter*, you have to inherit from *SpikeMetric*. Inputs of the metric in *get_features* are a nested list structure for the spikes generated by the model: a list where each element contains the results for a single parameter set. Each of these results is a list for each of the input traces, where the elements of this list are numpy arrays of spike times (without units, i.e. in seconds). For example, if two parameters sets and 3 different input stimuli were tested, this structure could look like this:

```
[
    [array([0.01, 0.5]), array([]), array([])],
    [array([0.02]), array([]), array([])]
]
```

This means that the both parameter sets only generate spikes for the first input stimulus, but the first parameter sets generates two while the second generates only a single one.

The target spikes are represented in the same way as a list of spike times for each input stimulus. The results of the function have to be returned as in *TraceMetric*, i.e. as a 2-d array of shape `(n_samples, n_traces)`.

2.4 Advanced Features

This part of documentation list other features provided alongside or inside *Fitter* objects, to help the user with easier and more flexible applications.

2.4.1 Parameters initialization

Whilst running *Fitter* user can specify values with which model evaluation of differential equations start.

The fitting functions accept additional dictionary input to address that. To do so, dictionary argument has to be added to *Fitter* initialization:

```
param_init = {'v': -30*mV}
```

```
fitter = TraceFitter(..., param_init = {'v': -30*mV})
```

2.4.2 Restart

By default any *Fitter* works in continuous optimization mode between run, where all of the parameters drawn are being evaluated.

Through changing the input flag in *fit()*: `restart` to `True`, user can reset the optimizer and start the optimization from scratch.

Used by Fitter optimizer and metric can only be changed when the flat is `True`.

2.4.3 Callback function

To visualize the progress of the optimization we provided few possibilities of feedback inside *Fitter*.

The 'callback' input provides few default options, updated in each round:

- `'text'` (default) that prints out the parameters of the best fit and corresponding error
- `'progressbar'` that uses `tqdm.autonotebook` to provide a progress bar
- `None` for non-verbose option

as well as **customized feedback option**. User can provide a *callable* (i.e. function), that will provide an output or printout. If callback returns `True` the fitting execution is interrupted.

User gets four arguments to customize over:

- `params` - set of parameters from current round
- `errors` - set of errors from current round
- `best_params` - best parameters globally, from all rounds
- `best_error` - best parameters globally, from all rounds
- `index` - index of current round

An example function:

```
def callback(params, errors, best_params, best_error, index):
   print('index {} errors minimum: {}'.format(index, min(errors)))
```

```
fitter = TraceFitter(...)
result, error  = fitter.fit(..., callback=...)
```

2.4.4 Generate Traces

With the same *Fitter* class user can also generate the traces with newly optimized parameters.

To simulate and visualize the traces or spikes for the parameters of choice. For a quick access to best fitted set of parameters Fitter classes provided ready to use functions:

- *generate_traces* inside *TraceFitter*
- *generate_spikes* inside *SpikeFitter*

Functions can be called after fitting in the following manner, without any input arguments:

```
fitter = TraceFitter(...)
results, error = fitter.fit(...)
traces = fitter.generate_traces()
```

```
fitter = SpikeFitter(...)
results, error = fitter.fit(...)
spikes = fitter.generate_spikes()
```

Custom generate

To create traces for other parameters, or generate traces after spike train fitting, user can call the - *generate* call, that takes in following arguments:

```
fitter.generate(params=None, output_var=None, param_init=None, level=0)
```

Where `params` is a dictionary of parameters for which the traces we generate. `output_var` provides an option to pick one or more variable for visualization. With `param_init`, user can define the initial values for differential equations. `level` allows for specification of namespace level from which we get the constant parameters of the model.

If `output_var` is the name of a single variable name (or the special name `'spikes'`), a single `Quantity` (for normal variables) or a list of spikes time arrays (for `'spikes'`) will be returned. If a list of names is provided, then the result is a dictionary with all the results.

```
fitter = TraceFitter(...)
results, error = fitter.fit(...)
traces = fitter.generate(output_var=['v', 'h', 'n', 'm'])
v_trace = traces['v']
h_trace = traces['h']
...
```

2.4.5 Results

Fitter class stores all of the parameters examined by the optimizer as well as the corresponding error. To retrieve them you can call the - *`results`*.

```
fitter = TraceFitter(...)
...
traces = fitter.generate_traces()
```

```
fitter = SpikeFitter(...)
...
results = fitter.results(format='dataframe')
```

Results can be returned in one of the following formats:

- `'list'` (default) returns a list of dictionaries with corresponding parameters (including units) and errors
- `'dict'` returns a dictionary of arrays with corresponding parameters (including units) and errors
- `'dataframe'` returns a `DataFrame` (without units)

The use of units (only relevant for formats `'list'` and `'dict'`) can be switched on or off with the `use_units` argument. If it is not specified, it will default to the value used during the initialization of the `Fitter` (which itself defaults to `True`).

Example output:

`'list'`:

```
[{'gl': 80.63365773 * nsiemens, 'g_kd': 66.00430921 * usiemens, 'g_na': 145.15634566␣
↪* usiemens, 'errors': 0.00019059452295872703},
 {'gl': 83.29319947 * nsiemens, 'g_kd': 168.75187749 * usiemens, 'g_na': 130.64547027␣
↪* usiemens, 'errors': 0.00021434415430605653},
 ...]
```

`'dict'`:

```
{'g_na': array([176.4472297 , 212.57019659, ...]) * usiemens,
 'g_kd': array([ 43.82344525,  54.35309635, ...]) * usiemens,
```

(continues on next page)

(continued from previous page)

```
 'gl': array([ 69.23559876, 134.68463669, ...]) * nsiemens,
 'errors': array([1.16788502, 0.5253008 , ...])}
```

'dataframe':

```
g_na            gl      g_kd    errors
0  0.000280  8.870238e-08  0.000047  0.521425
1  0.000192  1.121861e-07  0.000118  0.387140
...
```

2.4.6 Standalone mode

Just like with regular Brian script, modelfitting computations can be performed in `Runtime` mode (default) or `Standalone` mode. (https://brian2.readthedocs.io/en/stable/user/computation.html)

To enable this mode, add the following line after your Brian import, but before your simulation code:

```
set_device('cpp_standalone')
```

Important notes:

Warning: In standlone mode one script can not be used to contain multiple - *`Fitter`*, use separate scripts!

Note that the generation of traces or spikes via *`generate`* will always use runtime mode, even when the fitting procedure uses standalone mode.

2.4.7 OnlineTraceFitter

`OnlineTraceFitter` was created to work with long traces or big optimization. This *`Fitter`* uses online Mean Square Error as a metric. When *`fit()`* is called there is no need of specifying a metric, that is by default set to None. Instead the errors are calculated with use of brian's `run_regularly`, with each simulation.

```
fitter = OnlineTraceFitter(model=model,
                           input=inp_traces,
                           output=out_traces,
                           input_var='I',
                           output_var='v',
                           dt=0.1*ms,
                           n_samples=5)

result, error = fitter.fit(optimizer=optimizer,
                           n_rounds=1,
                           gl=[1e-8*siemens*cm**-2 * area, 1e-3*siemens*cm**-2 *␣
↪area],)
```

2.4.8 Reference the target values in the equations

A model can refer to the target output values within the equations. For example, if you are fitting a membrane potential trace v (i.e. `output_var='v'`), then the equations can refer to the target trace as `v_target`. This allows you for

example to add a coupling term like `coupling*(v_target - v)` to the equation for `v`, pulling the trajectory towards the correct solution.

2.5 Examples

2.5.1 Simple Examples

Following pieces of code show an example of Fitter class calls with possible inputs.

TraceFitter

```
n_opt = NevergradOptimizer(method='PSO')
metric = MSEMetric()

fitter = TraceFitter(model=model,
                     input_var='I',
                     output_var='v',
                     input=inp_trace,
                     dt=0.1*ms,
                     method='exponential_euler',
                     output=out_trace,
                     n_samples=5)

results, error = fitter.fit(optimizer=n_opt,
                            metric=metric,
                            callback='text',
                            n_rounds=1,
                            param_init={'v': -65*mV},
                            gl=[10*nS*cm**-2 * area, 1*mS*cm**-2 * area],
                            g_na=[1*mS*cm**-2 * area, 2000*mS*cm**-2 * area],
                            g_kd=[1*mS*cm**-2 * area, 1000*mS*cm**-2 * area])
```

SpikeFitter

```
n_opt = SkoptOptimizer('ET')
metric = GammaFactor(dt, delta=2*ms)

fitter = SpikeFitter(model=eqs,
                     input_var='I',
                     dt=0.1*ms,
                     input=inp_traces,
                     output=out_spikes,
                     n_samples=30,
                     threshold='v > -50*mV',
                     reset='v = -70*mV',
                     method='exponential_euler')

results, error = fitter.fit(n_rounds=2,
                  optimizer=n_opt,
                  metric=metric,
                  gL=[20*nS, 40*nS],
                  C = [0.5*nF, 1.5*nF])
```

2.5.2 Multirun of Hodgkin-Huxley

Here you can download the data: `input_traces` `output_traces`

```
import numpy as np
from brian2 import *
from brian2modelfitting import *
```

To load the data, use following code:

```
import pandas as pd
# Load Input and Output Data
df_inp_traces = pd.read_csv('input_traces_hh.csv')
df_out_traces = pd.read_csv('output_traces_hh.csv')

inp_traces = df_inp_traces.to_numpy()
inp_traces = inp_traces[:, 1:]

out_traces = df_out_traces.to_numpy()
out_traces = out_traces[:, 1:]
```

Then the multiple round optimization can be run with following code:

```
# Model Fitting
## Parameters
area = 20000*umetre**2
El = -65*mV
EK = -90*mV
ENa = 50*mV
VT = -63*mV
dt = 0.01*ms
defaultclock.dt = dt

## Modle Definition
eqs = Equations(
'''
dv/dt = (gl*(El-v) - g_na*(m*m*m)*h*(v-ENa) - g_kd*(n*n*n*n)*(v-EK) + I)/Cm : volt
dm/dt = 0.32*(mV**-1)*(13.*mV-v+VT)/
    (exp((13.*mV-v+VT)/(4.*mV))-1.)/ms*(1-m)-0.28*(mV**-1)*(v-VT-40.*mV)/
    (exp((v-VT-40.*mV)/(5.*mV))-1.)/ms*m : 1
dn/dt = 0.032*(mV**-1)*(15.*mV-v+VT)/
    (exp((15.*mV-v+VT)/(5.*mV))-1.)/ms*(1.-n)-.5*exp((10.*mV-v+VT)/(40.*mV))/ms*n : 1
dh/dt = 0.128*exp((17.*mV-v+VT)/(18.*mV))/ms*(1.-h)-4./(1+exp((40.*mV-v+VT)/(5.*mV)))/
↪ms*h : 1
g_na : siemens (constant)
g_kd : siemens (constant)
gl   : siemens (constant)
Cm   : farad (constant)
''')

## Optimization and Metric Choice
n_opt = NevergradOptimizer()
metric = MSEMetric()

## Fitting
fitter = TraceFitter(model=eqs, input_var='I', output_var='v',
                     input=inp_traces*amp, output=out_traces*mV, dt=dt,
                     n_samples=20,
```

(continues on next page)

(continued from previous page)

```
                    param_init={'v': -65*mV},
                    method='exponential_euler')

res, error = fitter.fit(n_rounds=2,
                        optimizer=n_opt, metric=metric,
                        callback='progressbar',
                        gl = [1e-09 *siemens, 1e-07 *siemens],
                        g_na = [2e-06*siemens, 2e-04*siemens],
                        g_kd = [6e-07*siemens, 6e-05*siemens],
                        Cm=[0.1*ufarad*cm**-2 * area, 2*ufarad*cm**-2 * area])

## Show results
all_output = fitter.results(format='dataframe')
print(all_output)

# Second round
res, error = fitter.fit(restart=True,
                        n_rounds=20,
                        optimizer=n_opt, metric=metric,
                        callback='progressbar',
                        gl = [1e-09 *siemens, 1e-07 *siemens],
                        g_na = [2e-06*siemens, 2e-04*siemens],
                        g_kd = [6e-07*siemens, 6e-05*siemens],
                        Cm=[0.1*ufarad*cm**-2 * area, 2*ufarad*cm**-2 * area])
```

To get the results and traces:

```
## Show results
all_output = fitter.results(format='dataframe')
print(all_output)

## Visualization of the results
fits = fitter.generate_traces(params=None, param_init={'v': -65*mV})

fig, axes = plt.subplots(ncols=5, figsize=(20,5), sharey=True)

for ax, data, fit in zip(axes, out_traces, fits):
    ax.plot(data.transpose())
    ax.plot(fit.transpose()/mV)

plt.show()
```

CHAPTER 3

API reference

3.1 brian2modelfitting package

3.1.1 Subpackages and -modules

brian2modelfitting.tests package

brian2modelfitting.tests.**run**()

brian2modelfitting.metric module

class brian2modelfitting.metric.**FeatureMetric**(*stim_times*, *feat_list*, *weights=None*, *combine=None*, *t_start=0. * second*, *normalization=1.0*)

Bases: *brian2modelfitting.metric.TraceMetric*

calc(*model_traces*, *data_traces*, *dt*)

Perform the error calculation across all parameters, calculate error between each output trace and corresponding simulation.

Parameters

- **model_traces** (ndarray) – Traces that should be evaluated and compared to the target data. Provided as an ndarray of shape (n_samples, n_traces, time steps) where n_samples is the number of parameter sets that have been evaluated, and n_traces is the number of stimuli.
- **data_traces** (ndarray) – The target traces to which the model should be compared. An ndarray of shape (n_traces, time steps).
- **dt** (Quantity) – The length of a single time step.

Returns Total error for each set of parameters, i.e. an array of shape (n_samples,).

Return type ndarray

check_values(*feat_list*)
Removes all the None values and checks for array features

feat_to_err(*d1*, *d2*)

get_dimensions(*output_dim*)
The physical dimensions of the error. In metrics such as *MSEMetric*, this depends on the dimensions of the output variable (e.g. if the output variable has units of volts, the mean squared error will have units of volt^2); in other metrics, e.g. *FeatureMetric*, this cannot be defined in a meaningful way since the metric combines different types of errors. In cases where defining dimensions is not meaningful, this method should return `DIMENSIONLESS`.

Parameters **output_dim** (`Dimension`) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the error.

Return type `Dimension`

get_errors(*features*)
Function weights features/multiple errors into one final error per each set of parameters.

The output of the function has to take shape of (n_samples,).

Parameters **features** (`ndarray`) – 2D array of shape `(n_samples, n_traces)` with the features/errors for each simulated trace

Returns Errors for each parameter set, i.e. of shape `(n_samples, )`

Return type `ndarray`

get_features(*traces*, *output*, *dt*)
Calculate the features/errors for each simulated trace, by comparing it to the corresponding data trace.

Parameters

- **model_traces** (`ndarray`) – Traces that should be evaluated and compared to the target data. Provided as an `ndarray` of shape `(n_samples, n_traces, time steps)`, where `n_samples` are the number of different parameter sets that have been evaluated, and `n_traces` are the number of input stimuli.
- **data_traces** (`ndarray`) – The target traces to which the model should be compared. An `ndarray` of shape `(n_traces, time steps)`.
- **dt** (`Quantity`) – The length of a single time step.

Returns An `ndarray` of shape `(n_samples, n_traces)` returning the error/feature value for each simulated trace.

Return type `ndarray`

get_normalized_dimensions(*output_dim*)
The physical dimensions of the normalized error. This will be the same as the dimensions returned by *get_dimensions* if the `normalization` is not used or set to a dimensionless value.

Parameters **output_dim** (`Dimension`) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the normalized error.

Return type `Dimension`

class brian2modelfitting.metric.**GammaFactor**(***kwds*)
Bases: *brian2modelfitting.metric.SpikeMetric*

Calculate gamma factors between goal and calculated spike trains, with precision delta.

Parameters

- **delta** (Quantity) – time window
- **time** (Quantity) – total length of experiment
- **rate_correction** (*bool*) – Whether to include an error term that penalizes differences in firing rate, following Clopath et al., Neurocomputing (2007). Defaults to True.

Notes

The gamma factor is commonly defined as 1 for a perfect match and 0 for a match not better than random (negative values are possible if the match is *worse* than expected by chance). Since we use the gamma factor as an error to be minimized, the calculated term is actually r - gamma_factor, where r is 1 if rate_correction is False, or a rate-difference dependent term if rate_correction` is ``True. In both cases, the best possible error value (i.e. for a perfect match between spike trains) is 0.

References

- R. Jolivet et al. "A Benchmark Test for a Quantitative Assessment of Simple Neuron Models." Journal of Neuroscience Methods, 169, no. 2 (2008): 417–24.
- C. Clopath et al. "Predicting Neuronal Activity with Simple Models of the Threshold Type: Adaptive Exponential Integrate-and-Fire Model with Two Compartments." Neurocomputing, 70, no. 10 (2007): 1668–73.

calc(*model_spikes*, *data_spikes*, *dt*)
Perform the error calculation across all parameters, calculate error between each output trace and corresponding simulation.

Parameters

- **model_spikes** (list of list of ndarray) – A nested list structure for the spikes generated by the model: a list where each element contains the results for a single parameter set. Each of these results is a list for each of the input traces, where the elements of this list are numpy arrays of spike times (without units, i.e. in seconds).
- **data_spikes** (list of ndarray) – The target spikes for the fitting, represented in the same way as model_spikes, i.e. as a list of spike times for each input stimulus.
- **dt** (Quantity) – The length of a single time step.

Returns Total error for each set of parameters, i.e. an array of shape (n_samples,)

Return type ndarray

get_dimensions(*output_dim*)
The physical dimensions of the error. In metrics such as *MSEMetric*, this depends on the dimensions of the output variable (e.g. if the output variable has units of volts, the mean squared error will have units of volt^2); in other metrics, e.g. *FeatureMetric*, this cannot be defined in a meaningful way since the metric combines different types of errors. In cases where defining dimensions is not meaningful, this method should return DIMENSIONLESS.

Parameters **output_dim** (Dimension) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the error.

Return type Dimension

get_errors(*features*)
Function weights features/multiple errors into one final error per each set of parameters.

The output of the function has to take shape of (n_samples,).

Parameters **features** (ndarray) – 2D array of shape (n_samples, n_traces) with the features/errors for each simulated trace

Returns Errors for each parameter set, i.e. of shape (n_samples,)

Return type ndarray

get_features(*traces*, *output*, *dt*)

Calculate the features/errors for each simulated spike train by comparing it to the corresponding data spike train.

Parameters

- **model_spikes** (list of list of ndarray) – A nested list structure for the spikes generated by the model: a list where each element contains the results for a single parameter set. Each of these results is a list for each of the input traces, where the elements of this list are numpy arrays of spike times (without units, i.e. in seconds).
- **data_spikes** (list of ndarray) – The target spikes for the fitting, represented in the same way as model_spikes, i.e. as a list of spike times for each input stimulus.
- **dt** (Quantity) – The length of a single time step.

Returns An ndarray of shape (n_samples, n_traces) returning the error/feature value for each simulated trace.

Return type ndarray

get_normalized_dimensions(*output_dim*)

The physical dimensions of the normalized error. This will be the same as the dimensions returned by *get_dimensions* if the normalization is not used or set to a dimensionless value.

Parameters **output_dim** (Dimension) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the normalized error.

Return type Dimension

class brian2modelfitting.metric.**MSEMetric**(***kwds*)

Bases: *brian2modelfitting.metric.TraceMetric*

Mean Square Error between goal and calculated output.

calc(*model_traces*, *data_traces*, *dt*)

Perform the error calculation across all parameters, calculate error between each output trace and corresponding simulation.

Parameters

- **model_traces** (ndarray) – Traces that should be evaluated and compared to the target data. Provided as an ndarray of shape (n_samples, n_traces, time steps) where n_samples is the number of parameter sets that have been evaluated, and n_traces is the number of stimuli.
- **data_traces** (ndarray) – The target traces to which the model should be compared. An ndarray of shape (n_traces, time steps).
- **dt** (Quantity) – The length of a single time step.

Returns Total error for each set of parameters, i.e. an array of shape (n_samples,).

Return type ndarray

get_dimensions(*output_dim*)
The physical dimensions of the error. In metrics such as *MSEMetric*, this depends on the dimensions of the output variable (e.g. if the output variable has units of volts, the mean squared error will have units of volt^2); in other metrics, e.g. *FeatureMetric*, this cannot be defined in a meaningful way since the metric combines different types of errors. In cases where defining dimensions is not meaningful, this method should return `DIMENSIONLESS`.

Parameters **output_dim** (`Dimension`) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the error.

Return type `Dimension`

get_errors(*features*)
Function weights features/multiple errors into one final error per each set of parameters.

The output of the function has to take shape of (n_samples,).

Parameters **features** (`ndarray`) – 2D array of shape `(n_samples, n_traces)` with the features/errors for each simulated trace

Returns Errors for each parameter set, i.e. of shape `(n_samples, )`

Return type `ndarray`

get_features(*model_traces*, *data_traces*, *dt*)
Calculate the features/errors for each simulated trace, by comparing it to the corresponding data trace.

Parameters

- **model_traces** (`ndarray`) – Traces that should be evaluated and compared to the target data. Provided as an `ndarray` of shape `(n_samples, n_traces, time steps)`, where `n_samples` are the number of different parameter sets that have been evaluated, and `n_traces` are the number of input stimuli.
- **data_traces** (`ndarray`) – The target traces to which the model should be compared. An `ndarray` of shape `(n_traces, time steps)`.
- **dt** (`Quantity`) – The length of a single time step.

Returns An `ndarray` of shape `(n_samples, n_traces)` returning the error/feature value for each simulated trace.

Return type `ndarray`

get_normalized_dimensions(*output_dim*)
The physical dimensions of the normalized error. This will be the same as the dimensions returned by *get_dimensions* if the `normalization` is not used or set to a dimensionless value.

Parameters **output_dim** (`Dimension`) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the normalized error.

Return type `Dimension`

class brian2modelfitting.metric.**Metric**(***kwds*)
Bases: `object`

Metric abstract class to define functions required for a custom metric To be used with modelfitting Fitters.

calc(*model_results*, *data_results*, *dt*)
Perform the error calculation across all parameter sets by comparing the simulated to the experimental data.

Parameters

- **model_results** – Results generated by the model. The type and shape of this data depends on the fitting problem. See *TraceMetric.calc* and *SpikeMetric.calc*.
- **data_results** – The experimental data that the model is fit against. See *TraceMetric.calc* and *SpikeMetric.calc* for the type/shape of the data.
- **dt** (Quantity) – The length of a single time step.

Returns Total error for each set of parameters, i.e. an array of shape `(n_samples, )`.

Return type ndarray

get_dimensions(*output_dim*)
The physical dimensions of the error. In metrics such as *MSEMetric*, this depends on the dimensions of the output variable (e.g. if the output variable has units of volts, the mean squared error will have units of volt^2); in other metrics, e.g. *FeatureMetric*, this cannot be defined in a meaningful way since the metric combines different types of errors. In cases where defining dimensions is not meaningful, this method should return `DIMENSIONLESS`.

Parameters **output_dim** (Dimension) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the error.

Return type Dimension

get_errors(*features*)
Function weights features/multiple errors into one final error per each set of parameters.

The output of the function has to take shape of (n_samples,).

Parameters **features** (ndarray) – 2D array of shape `(n_samples, n_traces)` with the features/errors for each simulated trace

Returns Errors for each parameter set, i.e. of shape `(n_samples, )`

Return type ndarray

get_features(*model_results*, *target_results*, *dt*)
Function calculates features / errors for each of the input traces.

The output of the function has to take shape of (n_samples, n_traces).

get_normalized_dimensions(*output_dim*)
The physical dimensions of the normalized error. This will be the same as the dimensions returned by *get_dimensions* if the `normalization` is not used or set to a dimensionless value.

Parameters **output_dim** (Dimension) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the normalized error.

Return type Dimension

class brian2modelfitting.metric.**SpikeMetric**(***kwds*)
Bases: *brian2modelfitting.metric.Metric*

A metric for comparing the spike trains.

calc(*model_spikes*, *data_spikes*, *dt*)
Perform the error calculation across all parameters, calculate error between each output trace and corresponding simulation.

Parameters

- **model_spikes** (list of list of ndarray) – A nested list structure for the spikes generated by the model: a list where each element contains the results for a single parameter

set. Each of these results is a list for each of the input traces, where the elements of this list are numpy arrays of spike times (without units, i.e. in seconds).

- **data_spikes** (list of ndarray) – The target spikes for the fitting, represented in the same way as model_spikes, i.e. as a list of spike times for each input stimulus.
- **dt** (Quantity) – The length of a single time step.

Returns Total error for each set of parameters, i.e. an array of shape (n_samples,)

Return type ndarray

get_dimensions(*output_dim*)

The physical dimensions of the error. In metrics such as *MSEMetric*, this depends on the dimensions of the output variable (e.g. if the output variable has units of volts, the mean squared error will have units of volt^2); in other metrics, e.g. *FeatureMetric*, this cannot be defined in a meaningful way since the metric combines different types of errors. In cases where defining dimensions is not meaningful, this method should return DIMENSIONLESS.

Parameters **output_dim** (Dimension) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the error.

Return type Dimension

get_errors(*features*)

Function weights features/multiple errors into one final error per each set of parameters.

The output of the function has to take shape of (n_samples,).

Parameters **features** (ndarray) – 2D array of shape (n_samples, n_traces) with the features/errors for each simulated trace

Returns Errors for each parameter set, i.e. of shape (n_samples,)

Return type ndarray

get_features(*model_spikes*, *data_spikes*, *dt*)

Calculate the features/errors for each simulated spike train by comparing it to the corresponding data spike train.

Parameters

- **model_spikes** (list of list of ndarray) – A nested list structure for the spikes generated by the model: a list where each element contains the results for a single parameter set. Each of these results is a list for each of the input traces, where the elements of this list are numpy arrays of spike times (without units, i.e. in seconds).
- **data_spikes** (list of ndarray) – The target spikes for the fitting, represented in the same way as model_spikes, i.e. as a list of spike times for each input stimulus.
- **dt** (Quantity) – The length of a single time step.

Returns An ndarray of shape (n_samples, n_traces) returning the error/feature value for each simulated trace.

Return type ndarray

get_normalized_dimensions(*output_dim*)

The physical dimensions of the normalized error. This will be the same as the dimensions returned by *get_dimensions* if the normalization is not used or set to a dimensionless value.

Parameters **output_dim** (Dimension) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the normalized error.

Return type Dimension

class brian2modelfitting.metric.**TraceMetric**(***kwds*)
Bases: *brian2modelfitting.metric.Metric*

Input traces have to be shaped into 2D array.

calc(*model_traces*, *data_traces*, *dt*)
Perform the error calculation across all parameters, calculate error between each output trace and corresponding simulation.

Parameters

- **model_traces** (ndarray) – Traces that should be evaluated and compared to the target data. Provided as an ndarray of shape (n_samples, n_traces, time steps) where n_samples is the number of parameter sets that have been evaluated, and n_traces is the number of stimuli.
- **data_traces** (ndarray) – The target traces to which the model should be compared. An ndarray of shape (n_traces, time steps).
- **dt** (Quantity) – The length of a single time step.

Returns Total error for each set of parameters, i.e. an array of shape (n_samples,).

Return type ndarray

get_dimensions(*output_dim*)
The physical dimensions of the error. In metrics such as *MSEMetric*, this depends on the dimensions of the output variable (e.g. if the output variable has units of volts, the mean squared error will have units of volt^2); in other metrics, e.g. *FeatureMetric*, this cannot be defined in a meaningful way since the metric combines different types of errors. In cases where defining dimensions is not meaningful, this method should return DIMENSIONLESS.

Parameters **output_dim** (Dimension) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the error.

Return type Dimension

get_errors(*features*)
Function weights features/multiple errors into one final error per each set of parameters.

The output of the function has to take shape of (n_samples,).

Parameters **features** (ndarray) – 2D array of shape (n_samples, n_traces) with the features/errors for each simulated trace

Returns Errors for each parameter set, i.e. of shape (n_samples,)

Return type ndarray

get_features(*model_traces*, *data_traces*, *dt*)
Calculate the features/errors for each simulated trace, by comparing it to the corresponding data trace.

Parameters

- **model_traces** (ndarray) – Traces that should be evaluated and compared to the target data. Provided as an ndarray of shape (n_samples, n_traces, time steps), where n_samples are the number of different parameter sets that have been evaluated, and n_traces are the number of input stimuli.
- **data_traces** (ndarray) – The target traces to which the model should be compared. An ndarray of shape (n_traces, time steps).

- **dt** (Quantity) – The length of a single time step.

Returns An ndarray of shape (n_samples, n_traces) returning the error/feature value for each simulated trace.

Return type ndarray

get_normalized_dimensions(*output_dim*)
The physical dimensions of the normalized error. This will be the same as the dimensions returned by *get_dimensions* if the normalization is not used or set to a dimensionless value.

Parameters **output_dim** (Dimension) – The dimensions of the output variable.

Returns **dim** – The physical dimensions of the normalized error.

Return type Dimension

brian2modelfitting.metric.**calc_eFEL**(*traces*, *inp_times*, *feat_list*, *dt*)

brian2modelfitting.metric.**firing_rate**(*spikes*)
Returns rate of the spike train

brian2modelfitting.metric.**get_gamma_factor**(*model*, *data*, *delta*, *time*, *dt*, *rate_correction=True*)
Calculate gamma factor between model and target spike trains, with precision delta.

Parameters

- **model** (list or ndarray) – model trace
- **data** (list or ndarray) – data trace
- **delta** (Quantity) – time window
- **dt** (Quantity) – time step
- **time** (Quantity) – total time of the simulation
- **rate_correction** (*bool*) – Whether to include an error term that penalizes differences in firing rate, following Clopath et al., Neurocomputing (2007).

Returns An error based on the Gamma factor. If rate_correction is used, then the returned error is $1 + 2\frac{|r_{\text{data}} - r_{\text{model}}|}{r_{\text{data}}} - \Gamma$ (with r_{data} and r_{model} being the firing rates in the data/model, and Γ the coincidence factor). Without rate_correction, the error is $1 - \Gamma$. Note that the coincidence factor Γ has a maximum value of 1 (when the two spike trains are exactly identical) and a value of 0 if there are only as many coincidences as expected from two homogeneous Poisson processes of the same rate. It can also take negative values if there are fewer coincidences than expected by chance.

Return type float

brian2modelfitting.metric.**normalize_weights**(*t_weights*)

brian2modelfitting.fitter module

class brian2modelfitting.fitter.**Fitter**(*dt*, *model*, *input*, *output*, *input_var*, *output_var*, *n_samples*, *threshold*, *reset*, *refractory*, *method*, *param_init*, *penalty*, *use_units=True*)
Bases: object

Base Fitter class for model fitting applications.

Creates an interface for model fitting of traces with parameters draw by gradient-free algorithms (through ask/tell interfaces).

Initiates n_neurons = num input traces * num samples, to which drawn parameters get assigned and evaluates them in parallel.

Parameters

- **dt** (Quantity) – The size of the time step.
- **model** (Equations or str) – The equations describing the model.
- **input** (ndarray or Quantity) – A 2D array of shape (n_traces, time steps) given the input that will be fed into the model.
- **output** (Quantity or list) – Recorded output of the model that the model should reproduce. Should be a 2D array of the same shape as the input when fitting traces with *TraceFitter*, a list of spike times when fitting spike trains with *SpikeFitter*.
- **input_var** (*str*) – The name of the input variable in the model. Note that this variable should be *used* in the model (e.g. a variable I that is added as a current in the membrane potential equation), but not *defined*.
- **output_var** (*str*) – The name of the output variable in the model. Only needed when fitting traces with *TraceFitter*.
- **n_samples** (*int*) – Number of parameter samples to be optimized over in a single iteration.
- **threshold** (str, optional) – The condition which produces spikes. Should be a boolean expression as a string.
- **reset** (str, optional) – The (possibly multi-line) string with the code to execute on reset.
- **refractory** (str or Quantity, optional) – Either the length of the refractory period (e.g. 2*ms), a string expression that evaluates to the length of the refractory period after each spike (e.g. '(1 + rand())*ms'), or a string expression evaluating to a boolean value, given the condition under which the neuron stays refractory after a spike (e.g. 'v > -20*mV')
- **method** (str, optional) – Integration method
- **penalty** (*str, optional*) – The name of a variable or subexpression in the model that will be added to the error at the end of each iteration. Note that only the term at the end of the simulation is taken into account, so this term should not be varying in time.
- **param_init** (dict, optional) – Dictionary of variables to be initialized with respective values

best_error

best_params

calc_errors(*metric*)

Abstract method required in all Fitter classes, used for calculating errors

Parameters **metric** (*Metric* children) – Child of Metric class, specifies optimization metric

fit(*optimizer*, *metric=None*, *n_rounds=1*, *callback='text'*, *restart=False*, *online_error=False*, *start_iteration=None*, *penalty=None*, *level=0*, ***params*)

Run the optimization algorithm for given amount of rounds with given number of samples drawn. Return best set of parameters and corresponding error.

Parameters

- **optimizer** (*Optimizer* children) – Child of Optimizer class, specific for each library.
- **metric** (*Metric* children) – Child of Metric class, specifies optimization metric

- **n_rounds** (*int*) – Number of rounds to optimize over (feedback provided over each round).
- **callback** (`str` or `Callable`) – Either the name of a provided callback function (`text` or `progressbar`), or a custom feedback function `func(parameters, errors, best_parameters, best_error, index)`. If this function returns `True` the fitting execution is interrupted.
- **restart** (*bool*) – Flag that reinitializes the Fitter to reset the optimization. With restart True user is allowed to change optimizer/metric.
- **online_error** (*bool, optional*) – Whether to calculate the squared error between target trace and simulated trace online. Defaults to `False`.
- **start_iteration** (*int, optional*) – A value for the `iteration` variable at the first iteration. If not given, will use 0 for the first call of `fit` (and for later calls when `restart` is specified). Later calls will continue to increase the value from the previous calls.
- **penalty** (*str, optional*) – The name of a variable or subexpression in the model that will be added to the error at the end of each iteration. Note that only the term at the end of the simulation is taken into account, so this term should not be varying in time. If not given, will reuse the value specified during `Fitter` initialization.
- **level** (`int`, optional) – How much farther to go down in the stack to find the namespace.
- ****params** – bounds for each parameter

Returns

- **best_results** (*dict*) – dictionary with best parameter set
- **error** (*float*) – error value for best parameter set

generate(*output_var=None*, *params=None*, *param_init=None*, *iteration=1000000000.0*, *level=0*)

Generates traces for best fit of parameters and all inputs. If provided with other parameters provides those.

Parameters

- **output_var** (*str or sequence of str*) – Name of the output variable to be monitored, or the special name `spikes` to record spikes. Can also be a sequence of names to record multiple variables.
- **params** (*dict*) – Dictionary of parameters to generate fits for.
- **param_init** (*dict*) – Dictionary of initial values for the model.
- **iteration** (*int, optional*) – Value for the `iteration` variable provided to the simulation. Defaults to a high value (1e9). This is based on the assumption that the model implements some coupling of the fitted variable to the target variable, and that this coupling inversely depends on the iteration number. In this case, one would usually want to switch off the coupling when generating traces/spikes for given parameters.
- **level** (`int`, optional) – How much farther to go down in the stack to find the namespace.

Returns Either a 2D `Quantity` with the recorded output variable over time, with shape <number of input traces> × <number of time steps>, or a list of spike times for each input trace. If several names were given as `output_var`, then the result is a dictionary with the names of the variable as the key.

Return type fit

optimization_iter(*optimizer*, *metric*, *penalty*)

Function performs all operations required for one iteration of optimization. Drawing parameters, setting them to simulator and calulating the error.

Parameters

- **optimizer** (Optimizer) –
- **metric** (Metric) –
- **penalty** (*str, optional*) – The name of a variable or subexpression in the model that will be added to the error at the end of each iteration. Note that only the term at the end of the simulation is taken into account, so this term should not be varying in time.

Returns

- **results** (*list*) – recommended parameters
- **parameters** (*list of list*) – drawn parameters
- **errors** (*list*) – calculated errors

results(*format='list'*, *use_units=None*)

Returns all of the gathered results (parameters and errors). In one of the 3 formats: 'dataframe', 'list', 'dict'.

Parameters

- **format** (*str*) – The desired output format. Currently supported: `dataframe`, `list`, or `dict`.
- **use_units** (*bool, optional*) – Whether to use units in the results. If not specified, defaults to `Tracefitter.use_units`, i.e. the value that was specified when the `Tracefitter` object was created (`True` by default).

Returns 'dataframe': returns pandas `DataFrame` without units 'list': list of dictionaries 'dict': dictionary of lists

Return type object

setup_neuron_group(*n_neurons*, *namespace*, *calc_gradient=False*, *optimize=True*, *online_error=False*, *name='neurons'*)

Setup neuron group, initialize required number of neurons, create namespace and initialize the parameters.

Parameters

- **n_neurons** (*int*) – number of required neurons
- ****namespace** – arguments to be added to NeuronGroup namespace

Returns **neurons** – group of neurons

Return type NeuronGroup

setup_simulator(*network_name*, *n_neurons*, *output_var*, *param_init*, *calc_gradient=False*, *optimize=True*, *online_error=False*, *level=1*)

class brian2modelfitting.fitter.**OnlineTraceFitter**(*model*, *input_var*, *input*, *output_var*, *output*, *dt*, *n_samples=30*, *method=None*, *reset=None*, *refractory=False*, *threshold=None*, *param_init=None*, *t_start=0. * second*, *penalty=None*)

Bases: *brian2modelfitting.fitter.Fitter*

best_error

best_params

calc_errors(*metric=None*)
Calculates error in online fashion.To be used inside optim_iter.

fit(*optimizer*, *n_rounds=1*, *callback='text'*, *restart=False*, *start_iteration=None*, *penalty=None*, *level=0*, ***params*)
Run the optimization algorithm for given amount of rounds with given number of samples drawn. Return best set of parameters and corresponding error.

Parameters

- **optimizer** (*Optimizer* children) – Child of Optimizer class, specific for each library.
- **metric** (*Metric* children) – Child of Metric class, specifies optimization metric
- **n_rounds** (*int*) – Number of rounds to optimize over (feedback provided over each round).
- **callback** (`str` or `Callable`) – Either the name of a provided callback function (`text` or `progressbar`), or a custom feedback function `func(parameters, errors, best_parameters, best_error, index)`. If this function returns `True` the fitting execution is interrupted.
- **restart** (*bool*) – Flag that reinitializes the Fitter to reset the optimization. With restart True user is allowed to change optimizer/metric.
- **online_error** (*bool, optional*) – Whether to calculate the squared error between target trace and simulated trace online. Defaults to `False`.
- **start_iteration** (*int, optional*) – A value for the `iteration` variable at the first iteration. If not given, will use 0 for the first call of `fit` (and for later calls when `restart` is specified). Later calls will continue to increase the value from the previous calls.
- **penalty** (*str, optional*) – The name of a variable or subexpression in the model that will be added to the error at the end of each iteration. Note that only the term at the end of the simulation is taken into account, so this term should not be varying in time. If not given, will reuse the value specified during `Fitter` initialization.
- **level** (`int`, optional) – How much farther to go down in the stack to find the namespace.
- ****params** – bounds for each parameter

Returns

- **best_results** (*dict*) – dictionary with best parameter set
- **error** (*float*) – error value for best parameter set

generate(*output_var=None*, *params=None*, *param_init=None*, *iteration=1000000000.0*, *level=0*)
Generates traces for best fit of parameters and all inputs. If provided with other parameters provides those.

Parameters

- **output_var** (*str or sequence of str*) – Name of the output variable to be monitored, or the special name `spikes` to record spikes. Can also be a sequence of names to record multiple variables.
- **params** (*dict*) – Dictionary of parameters to generate fits for.
- **param_init** (*dict*) – Dictionary of initial values for the model.
- **iteration** (*int, optional*) – Value for the `iteration` variable provided to the simulation. Defaults to a high value (1e9). This is based on the assumption that the

model implements some coupling of the fitted variable to the target variable, and that this coupling inversely depends on the iteration number. In this case, one would usually want to switch off the coupling when generating traces/spikes for given parameters.

- **level** (int, optional) – How much farther to go down in the stack to find the namespace.

Returns Either a 2D Quantity with the recorded output variable over time, with shape <number of input traces> × <number of time steps>, or a list of spike times for each input trace. If several names were given as output_var, then the result is a dictionary with the names of the variable as the key.

Return type fit

generate_traces(*params=None*, *param_init=None*, *level=0*)
Generates traces for best fit of parameters and all inputs

optimization_iter(*optimizer*, *metric*, *penalty*)
Function performs all operations required for one iteration of optimization. Drawing parameters, setting them to simulator and calulating the error.

Parameters

- **optimizer** (Optimizer) –
- **metric** (Metric) –
- **penalty** (*str, optional*) – The name of a variable or subexpression in the model that will be added to the error at the end of each iteration. Note that only the term at the end of the simulation is taken into account, so this term should not be varying in time.

Returns

- **results** (*list*) – recommended parameters
- **parameters** (*list of list*) – drawn parameters
- **errors** (*list*) – calculated errors

results(*format='list'*, *use_units=None*)
Returns all of the gathered results (parameters and errors). In one of the 3 formats: 'dataframe', 'list', 'dict'.

Parameters

- **format** (*str*) – The desired output format. Currently supported: dataframe, list, or dict.
- **use_units** (*bool, optional*) – Whether to use units in the results. If not specified, defaults to Tracefitter.use_units, i.e. the value that was specified when the Tracefitter object was created (True by default).

Returns 'dataframe': returns pandas DataFrame without units 'list': list of dictionaries 'dict': dictionary of lists

Return type object

setup_neuron_group(*n_neurons*, *namespace*, *calc_gradient=False*, *optimize=True*, *online_error=False*, *name='neurons'*)
Setup neuron group, initialize required number of neurons, create namespace and initialize the parameters.

Parameters

- **n_neurons** (*int*) – number of required neurons
- ****namespace** – arguments to be added to NeuronGroup namespace

Returns **neurons** – group of neurons

Return type NeuronGroup

setup_simulator(*network_name*, *n_neurons*, *output_var*, *param_init*, *calc_gradient=False*, *optimize=True*, *online_error=False*, *level=1*)

class brian2modelfitting.fitter.**SpikeFitter**(*model*, *input*, *output*, *dt*, *reset*, *threshold*, *input_var='I'*, *refractory=False*, *n_samples=30*, *method=None*, *param_init=None*, *penalty=None*, *use_units=True*)

Bases: *brian2modelfitting.fitter.Fitter*

best_error

best_params

calc_errors(*metric*)
Returns errors after simulation with SpikeMonitor. To be used inside optim_iter.

fit(*optimizer*, *metric=None*, *n_rounds=1*, *callback='text'*, *restart=False*, *start_iteration=None*, *penalty=None*, *level=0*, ***params*)
Run the optimization algorithm for given amount of rounds with given number of samples drawn. Return best set of parameters and corresponding error.

Parameters

- **optimizer** (*Optimizer* children) – Child of Optimizer class, specific for each library.
- **metric** (*Metric* children) – Child of Metric class, specifies optimization metric
- **n_rounds** (*int*) – Number of rounds to optimize over (feedback provided over each round).
- **callback** (`str` or `Callable`) – Either the name of a provided callback function (`text` or `progressbar`), or a custom feedback function `func(parameters, errors, best_parameters, best_error, index)`. If this function returns `True` the fitting execution is interrupted.
- **restart** (*bool*) – Flag that reinitializes the Fitter to reset the optimization. With restart True user is allowed to change optimizer/metric.
- **online_error** (*bool, optional*) – Whether to calculate the squared error between target trace and simulated trace online. Defaults to `False`.
- **start_iteration** (*int, optional*) – A value for the `iteration` variable at the first iteration. If not given, will use 0 for the first call of `fit` (and for later calls when `restart` is specified). Later calls will continue to increase the value from the previous calls.
- **penalty** (*str, optional*) – The name of a variable or subexpression in the model that will be added to the error at the end of each iteration. Note that only the term at the end of the simulation is taken into account, so this term should not be varying in time. If not given, will reuse the value specified during `Fitter` initialization.
- **level** (`int`, optional) – How much farther to go down in the stack to find the namespace.
- ****params** – bounds for each parameter

Returns

- **best_results** (*dict*) – dictionary with best parameter set
- **error** (*float*) – error value for best parameter set

generate (*output_var=None*, *params=None*, *param_init=None*, *iteration=1000000000.0*, *level=0*)
Generates traces for best fit of parameters and all inputs. If provided with other parameters provides those.

Parameters

- **output_var** (*str or sequence of str*) – Name of the output variable to be monitored, or the special name `spikes` to record spikes. Can also be a sequence of names to record multiple variables.
- **params** (*dict*) – Dictionary of parameters to generate fits for.
- **param_init** (*dict*) – Dictionary of initial values for the model.
- **iteration** (*int, optional*) – Value for the `iteration` variable provided to the simulation. Defaults to a high value (1e9). This is based on the assumption that the model implements some coupling of the fitted variable to the target variable, and that this coupling inversely depends on the iteration number. In this case, one would usually want to switch off the coupling when generating traces/spikes for given parameters.
- **level** (`int`, optional) – How much farther to go down in the stack to find the namespace.

Returns Either a 2D `Quantity` with the recorded output variable over time, with shape <number of input traces> × <number of time steps>, or a list of spike times for each input trace. If several names were given as `output_var`, then the result is a dictionary with the names of the variable as the key.

Return type fit

generate_spikes (*params=None*, *param_init=None*, *iteration=1000000000.0*, *level=0*)
Generates traces for best fit of parameters and all inputs

optimization_iter (*optimizer*, *metric*, *penalty*)
Function performs all operations required for one iteration of optimization. Drawing parameters, setting them to simulator and calulating the error.

Parameters

- **optimizer** (`Optimizer`) –
- **metric** (`Metric`) –
- **penalty** (*str, optional*) – The name of a variable or subexpression in the model that will be added to the error at the end of each iteration. Note that only the term at the end of the simulation is taken into account, so this term should not be varying in time.

Returns

- **results** (*list*) – recommended parameters
- **parameters** (*list of list*) – drawn parameters
- **errors** (*list*) – calculated errors

results (*format='list'*, *use_units=None*)
Returns all of the gathered results (parameters and errors). In one of the 3 formats: 'dataframe', 'list', 'dict'.

Parameters

- **format** (*str*) – The desired output format. Currently supported: `dataframe`, `list`, or `dict`.
- **use_units** (*bool, optional*) – Whether to use units in the results. If not specified, defaults to `Tracefitter.use_units`, i.e. the value that was specified when the `Tracefitter` object was created (`True` by default).

Returns 'dataframe': returns pandas `DataFrame` without units 'list': list of dictionaries 'dict': dictionary of lists

Return type object

setup_neuron_group(*n_neurons*, *namespace*, *calc_gradient=False*, *optimize=True*, *online_error=False*, *name='neurons'*)

Setup neuron group, initialize required number of neurons, create namespace and initialize the parameters.

Parameters

- **n_neurons** (*int*) – number of required neurons
- ****namespace** – arguments to be added to NeuronGroup namespace

Returns **neurons** – group of neurons

Return type NeuronGroup

setup_simulator(*network_name*, *n_neurons*, *output_var*, *param_init*, *calc_gradient=False*, *optimize=True*, *online_error=False*, *level=1*)

class brian2modelfitting.fitter.**TraceFitter**(*model*, *input_var*, *input*, *output_var*, *output*, *dt*, *n_samples=30*, *method=None*, *reset=None*, *refractory=False*, *threshold=None*, *param_init=None*, *penalty=None*, *use_units=True*)

Bases: *brian2modelfitting.fitter.Fitter*

A *Fitter* for fitting recorded traces (e.g. of the membrane potential).

Parameters

- **model** –
- **input_var** –
- **input** –
- **output_var** –
- **output** –
- **dt** –
- **n_samples** –
- **method** –
- **reset** –
- **refractory** –
- **threshold** –
- **param_init** –
- **use_units** (*bool, optional*) – Whether to use units in all user-facing interfaces, e.g. in the callback arguments or in the returned parameter dictionary and errors. Defaults to `True`.

best_error

best_params

calc_errors(*metric*)

Returns errors after simulation with StateMonitor. To be used inside `optim_iter`.

fit(*optimizer*, *metric=None*, *n_rounds=1*, *callback='text'*, *restart=False*, *start_iteration=None*, *penalty=None*, *level=0*, ***params*)

Run the optimization algorithm for given amount of rounds with given number of samples drawn. Return best set of parameters and corresponding error.

Parameters

- **optimizer** (*Optimizer* children) – Child of Optimizer class, specific for each library.
- **metric** (*Metric* children) – Child of Metric class, specifies optimization metric
- **n_rounds** (*int*) – Number of rounds to optimize over (feedback provided over each round).
- **callback** (`str` or `Callable`) – Either the name of a provided callback function (`text` or `progressbar`), or a custom feedback function `func(parameters, errors, best_parameters, best_error, index)`. If this function returns `True` the fitting execution is interrupted.
- **restart** (*bool*) – Flag that reinitializes the Fitter to reset the optimization. With restart True user is allowed to change optimizer/metric.
- **online_error** (*bool, optional*) – Whether to calculate the squared error between target trace and simulated trace online. Defaults to `False`.
- **start_iteration** (*int, optional*) – A value for the `iteration` variable at the first iteration. If not given, will use 0 for the first call of `fit` (and for later calls when `restart` is specified). Later calls will continue to increase the value from the previous calls.
- **penalty** (*str, optional*) – The name of a variable or subexpression in the model that will be added to the error at the end of each iteration. Note that only the term at the end of the simulation is taken into account, so this term should not be varying in time. If not given, will reuse the value specified during `Fitter` initialization.
- **level** (`int`, optional) – How much farther to go down in the stack to find the namespace.
- ****params** – bounds for each parameter

Returns

- **best_results** (*dict*) – dictionary with best parameter set
- **error** (*float*) – error value for best parameter set

generate(*output_var=None*, *params=None*, *param_init=None*, *iteration=1000000000.0*, *level=0*)

Generates traces for best fit of parameters and all inputs. If provided with other parameters provides those.

Parameters

- **output_var** (*str or sequence of str*) – Name of the output variable to be monitored, or the special name `spikes` to record spikes. Can also be a sequence of names to record multiple variables.
- **params** (*dict*) – Dictionary of parameters to generate fits for.
- **param_init** (*dict*) – Dictionary of initial values for the model.
- **iteration** (*int, optional*) – Value for the `iteration` variable provided to the simulation. Defaults to a high value (1e9). This is based on the assumption that the model implements some coupling of the fitted variable to the target variable, and that this coupling inversely depends on the iteration number. In this case, one would usually want to switch off the coupling when generating traces/spikes for given parameters.
- **level** (`int`, optional) – How much farther to go down in the stack to find the namespace.

Returns Either a 2D `Quantity` with the recorded output variable over time, with shape <number of input traces> × <number of time steps>, or a list of spike times for each input trace. If several names were given as `output_var`, then the result is a dictionary with the names of the variable as the key.

Return type fit

generate_traces(*params=None*, *param_init=None*, *iteration=1000000000.0*, *level=0*)
Generates traces for best fit of parameters and all inputs

optimization_iter(*optimizer*, *metric*, *penalty*)
Function performs all operations required for one iteration of optimization. Drawing parameters, setting them to simulator and calulating the error.

Parameters

- **optimizer** (`Optimizer`) –
- **metric** (`Metric`) –
- **penalty** (*str, optional*) – The name of a variable or subexpression in the model that will be added to the error at the end of each iteration. Note that only the term at the end of the simulation is taken into account, so this term should not be varying in time.

Returns

- **results** (*list*) – recommended parameters
- **parameters** (*list of list*) – drawn parameters
- **errors** (*list*) – calculated errors

refine(*params=None*, *t_start=None*, *t_weights=None*, *normalization=None*, *callback='text'*, *calc_gradient=False*, *optimize=True*, *iteration=1000000000.0*, *level=0*, ***kwds*)
Refine the fitting results with a sequentially operating minimization algorithm. Uses the lmfit package which itself makes use of scipy.optimize. Has to be called after *fit*, but a call with `n_rounds=0` is enough.

Parameters

- **params** (*dict, optional*) – A dictionary with the parameters to use as a starting point for the refinement. If not given, the best parameters found so far by *fit* will be used.
- **t_start** (`Quantity`, optional) – Start of time window considered for calculating the fit error. If not set, will reuse the `t_start` value from the previously used metric.
- **t_weights** (`ndarray`, optional) – A 1-dimensional array of weights for each time point. This array has to have the same size as the input/output traces that are used for fitting. A value of 0 means that data points are ignored. The weight values will be normalized so only the relative values matter. For example, an array containing 1s, and 2s, will weigh the regions with 2s twice as high (with respect to the squared error) as the regions with 1s. Using instead values of 0.5 and 1 would have the same effect. Cannot be combined with `t_start`. If not set, will reuse the `t_weights` value from the previously used metric.
- **normalization** (*float, optional*) – A normalization term that will be used rescale results before handing them to the optimization algorithm. Can be useful if the algorithm makes assumptions about the scale of errors, e.g. if the size of steps in the parameter space depends on the absolute value of the error. The difference between simulated and target traces will be divided by this value. If not set, will reuse the `normalization` value from the previously used metric.

- **callback** (str or Callable) – Either the name of a provided callback function (text or progressbar), or a custom feedback function func(parameters, errors, best_parameters, best_error, index). If this function returns True the fitting execution is interrupted.
- **calc_gradient** (*bool, optional*) – Whether to add "sensitivity variables" to the equation that track the sensitivity of the equation variables to the parameters. This information will be used to pass the local gradient of the error with respect to the parameters to the optimization function. This can lead to much faster convergence than with an estimated gradient but comes at the expense of additional computation. Defaults to False.
- **optimize** (*bool, optional*) – Whether to remove sensitivity variables from the equations that do not evolve if initialized to zero (e.g. dS_x_y/dt = -S_x_y/tau would be removed). This avoids unnecessary computation but will fail in the rare case that such a sensitivity variable needs to be initialized to a non-zero value. Only taken into account if calc_gradient is True. Defaults to True.
- **iteration** (*int, optional*) – Value for the iteration variable provided to the simulation. Defaults to a high value (1e9). This is based on the assumption that the model implements some coupling of the fitted variable to the target variable, and that this coupling inversely depends on the iteration number. In this case, one would usually want to switch off the coupling when refining the solution.
- **level** (*int, optional*) – How much farther to go down in the stack to find the namespace.
- **kwds** – Additional arguments can overwrite the bounds for individual parameters (if not given, the bounds previously specified in the call to *fit* will be used). All other arguments will be passed on to lmfit.minimize and can be used to e.g. change the method, or to specify method-specific arguments.

Returns

- **parameters** (*dict*) – The parameters at the end of the optimization process as a dictionary.
- **result** (lmfit.MinimizerResult) – The result of the optimization process.

Notes

The default method used by lmfit is least-squares minimization using a Levenberg-Marquardt method. Note that there is no support for specifying a Metric, the given output trace(s) will be subtracted from the simulated trace(s) and passed on to the minimization algorithm.

This method always uses the runtime mode, independent of the selection of the current device.

results (*format='list', use_units=None*)

Returns all of the gathered results (parameters and errors). In one of the 3 formats: 'dataframe', 'list', 'dict'.

Parameters

- **format** (*str*) – The desired output format. Currently supported: dataframe, list, or dict.
- **use_units** (*bool, optional*) – Whether to use units in the results. If not specified, defaults to Tracefitter.use_units, i.e. the value that was specified when the Tracefitter object was created (True by default).

Returns 'dataframe': returns pandas DataFrame without units 'list': list of dictionaries 'dict': dictionary of lists

Return type object

setup_neuron_group(*n_neurons*, *namespace*, *calc_gradient=False*, *optimize=True*, *online_error=False*, *name='neurons'*)

Setup neuron group, initialize required number of neurons, create namespace and initialize the parameters.

Parameters

- **n_neurons** (*int*) – number of required neurons
- ****namespace** – arguments to be added to NeuronGroup namespace

Returns **neurons** – group of neurons

Return type NeuronGroup

setup_simulator(*network_name*, *n_neurons*, *output_var*, *param_init*, *calc_gradient=False*, *optimize=True*, *online_error=False*, *level=1*)

brian2modelfitting.fitter.**get_full_namespace**(*additional_namespace*, *level=0*)

brian2modelfitting.fitter.**get_param_dic**(*params*, *param_names*, *n_traces*, *n_samples*)

Transform parameters into a dictionary of appropiate size

brian2modelfitting.fitter.**get_sensitivity_equations**(*group*, *parameters*, *namespace=None*, *level=1*, *optimize=True*)

Get equations for sensitivity variables.

Parameters

- **group** (NeuronGroup) – The group of neurons that will be simulated.
- **parameters** (*list of str*) – Names of the parameters that are fit.
- **namespace** (*dict, optional*) – The namespace to use.
- **level** (int, optional) – How much farther to go down in the stack to find the namespace.
- **optimize** (*bool, optional*) – Whether to remove sensitivity variables from the equations that do not evolve if initialized to zero (e.g. `dS_x_y/dt = -S_x_y/tau` would be removed). This avoids unnecessary computation but will fail in the rare case that such a sensitivity variable needs to be initialized to a non-zero value. Defaults to `True`.

Returns **sensitivity_eqs** – The equations for the sensitivity variables.

Return type Equations

brian2modelfitting.fitter.**get_sensitivity_init**(*group*, *parameters*, *param_init*)

Calculate the initial values for the sensitivity parameters (necessary if initial values are functions of parameters).

Parameters

- **group** (NeuronGroup) – The group of neurons that will be simulated.
- **parameters** (*list of str*) – Names of the parameters that are fit.
- **param_init** (*dict*) – The dictionary with expressions to initialize the model variables.

Returns **sensitivity_init** – Dictionary of expressions to initialize the sensitivity parameters.

Return type dict

brian2modelfitting.fitter.**get_spikes**(*monitor*, *n_samples*, *n_traces*)

Get spikes from spike monitor change format from dict to a list, remove units.

brian2modelfitting.fitter.**setup_fit**()

Function sets up simulator in one of the two available modes: runtime or standalone. The *Simulator* that will be used depends on the currently set Device. In the case of CPPStandaloneDevice, the device will also be reset if it has already run a simulation.

Returns **simulator**

Return type *Simulator*

brian2modelfitting.optimizer module

class brian2modelfitting.optimizer.**NevergradOptimizer**(*method='DE'*, *use_nevergrad_recommendation=False*, ***kwds*)

Bases: *brian2modelfitting.optimizer.Optimizer*

NevergradOptimizer instance creates all the tools necessary for the user to use it with Nevergrad library.

Parameters

- **parameter_names** (list or dict) – List/Dict of strings with parameters to be used as instruments.
- **bounds** (list) – List with appropriate bounds for each parameter.
- **method** (str, optional) – The optimization method. By default differential evolution, can be chosen from any method in Nevergrad registry
- **use_nevergrad_recommendation** (*bool, optional*) – Whether to use Nevergrad's recommendation as the "best result". This recommendation takes several evaluations of the same parameters (for stochastic simulations) into account. The alternative is to simply return the parameters with the lowest error so far (the default). The problem with Nevergrad's recommendation is that it can give wrong result for errors that are very close in magnitude due (see github issue #16).
- **budget** (*int or None*) – number of allowed evaluations
- **num_workers** (*int*) – number of evaluations which will be run in parallel at once

ask(*n_samples*)

Returns the requested number of samples of parameter sets

Parameters **n_samples** (*int*) – number of samples to be drawn

Returns **parameters** – list of drawn parameters [n_samples x n_params]

Return type list

initialize(*parameter_names*, *popsize*, ***params*)

Initialize the instrumentation for the optimization, based on parameters, creates bounds for variables and attaches them to the optimizer

Parameters

- **parameter_names** (*list[str]*) – list of parameter names in use
- **popsize** (*int*) – population size
- ****params** – bounds for each parameter

recommend()

Returns best recommendation provided by the method

Returns **result** – list of best fit parameters[n_params]

Return type list

tell(*parameters*, *errors*)
Provides the evaluated errors from parameter sets to optimizer

Parameters

- **parameters** (*list*) – list of parameters [n_samples x n_params]
- **errors** (*list*) – list of errors [n_samples]

class brian2modelfitting.optimizer.**Optimizer**
Bases: object

Optimizer class created as a base for optimization initialization and performance with different libraries. To be used with modelfitting Fitter.

ask(*n_samples*)
Returns the requested number of samples of parameter sets

Parameters **n_samples** (*int*) – number of samples to be drawn

Returns **parameters** – list of drawn parameters [n_samples x n_params]

Return type list

initialize(*parameter_names*, *popsize*, ***params*)
Initialize the instrumentation for the optimization, based on parameters, creates bounds for variables and attaches them to the optimizer

Parameters

- **parameter_names** (*list[str]*) – list of parameter names in use
- **popsize** (*int*) – population size
- ****params** – bounds for each parameter

recommend()
Returns best recommendation provided by the method

Returns **result** – list of best fit parameters[n_params]

Return type list

tell(*parameters*, *errors*)
Provides the evaluated errors from parameter sets to optimizer

Parameters

- **parameters** (*list*) – list of parameters [n_samples x n_params]
- **errors** (*list*) – list of errors [n_samples]

class brian2modelfitting.optimizer.**SkoptOptimizer**(*method='GP'*, ***kwds*)
Bases: *brian2modelfitting.optimizer.Optimizer*

SkoptOptimizer instance creates all the tools necessary for the user to use it with scikit-optimize library.

Parameters

- **parameter_names** (*list[str]*) – Parameters to be used as instruments.
- **bounds** (*list*) – List with appropiate bounds for each parameter.
- **method** (str, optional) – The optimization method. Possibilities: “GP”, “RF”, “ET”, “GBRT” or sklearn regressor, default=”GP”

- **n_calls** (int) – Number of calls to func. Defaults to 100.

ask(*n_samples*)
Returns the requested number of samples of parameter sets

Parameters **n_samples** (*int*) – number of samples to be drawn

Returns **parameters** – list of drawn parameters [n_samples x n_params]

Return type list

initialize(*parameter_names*, *popsize*, ***params*)
Initialize the instrumentation for the optimization, based on parameters, creates bounds for variables and attaches them to the optimizer

Parameters

- **parameter_names** (*list[str]*) – list of parameter names in use
- **popsize** (*int*) – population size
- ****params** – bounds for each parameter

recommend()
Returns best recommendation provided by the method

Returns **result** – list of best fit parameters[n_params]

Return type list

tell(*parameters*, *errors*)
Provides the evaluated errors from parameter sets to optimizer

Parameters

- **parameters** (*list*) – list of parameters [n_samples x n_params]
- **errors** (*list*) – list of errors [n_samples]

brian2modelfitting.optimizer.**calc_bounds**(*parameter_names*, ***params*)
Verify and get the provided for parameters bounds

Parameters

- **parameter_names** (*list[str]*) – list of parameter names in use
- ****params** – bounds for each parameter

brian2modelfitting.simulator module

class brian2modelfitting.simulator.**CPPStandaloneSimulator**
Bases: *brian2modelfitting.simulator.Simulator*

Simulation class created for use with CPPStandaloneDevice

initialize(*network*, *var_init*, *name='fit'*)
Prepares the simulation for running

Parameters

- **network** (Network) – Network consisting of a NeuronGroup named neurons and either a monitor named spikemonitor or a monitor named ‘‘statemonitor‘‘(or both).
- **var_init** (*dict*) – dictionary to initialize the variable states
- **name** (str, optional) – name of the network

neurons

run (*duration*, *params*, *params_names*, *iteration*, *name='fit'*)
Simulation has to be run in two stages in order to initialize the code generation

spikemonitor

statemonitor

class brian2modelfitting.simulator.**RuntimeSimulator**
Bases: *brian2modelfitting.simulator.Simulator*

Simulation class created for use with RuntimeDevice

initialize (*network*, *var_init*, *name='fit'*)
Prepares the simulation for running

Parameters

- **network** (Network) – Network consisting of a NeuronGroup named neurons and either a monitor named spikemonitor or a monitor named ‘‘statemonitor‘‘(or both).
- **var_init** (*dict*) – dictionary to initialize the variable states
- **name** (str, optional) – name of the network

neurons

run (*duration*, *params*, *params_names*, *iteration*, *name='fit'*)
Restores the network, sets neurons to required parameters and runs the simulation

Parameters

- **duration** (Quantity) – Simulation duration
- **params** (*dict*) – parameters to be set
- **params_names** (*list[str]*) – names of parameters to set the dictionary

spikemonitor

statemonitor

class brian2modelfitting.simulator.**Simulator**
Bases: object

Simulation class created to perform a simulation for fitting traces or spikes.

initialize (*network*, *var_init*, *name='fit'*)
Prepares the simulation for running

Parameters

- **network** (Network) – Network consisting of a NeuronGroup named neurons and either a monitor named spikemonitor or a monitor named ‘‘statemonitor‘‘(or both).
- **var_init** (*dict*) – dictionary to initialize the variable states
- **name** (str, optional) – name of the network

neurons

run (*duration*, *params*, *params_names*, *iteration*, *name*)
Restores the network, sets neurons to required parameters and runs the simulation

Parameters

- **duration** (Quantity) – Simulation duration

- **params** (*dict*) – parameters to be set
- **params_names** (*list[str]*) – names of parameters to set the dictionary

spikemonitor

statemonitor

brian2modelfitting.simulator.**initialize_neurons**(*params_names*, *neurons*, *params*)
initialize each parameter for NeuronGroup returns dictionary of Dummy devices

brian2modelfitting.simulator.**initialize_parameter**(*variableview*, *value*)
initialize parameter variable in static file, returns Dummy device

brian2modelfitting.simulator.**run_again**()
re-run the NeuronGroup on cpp file

brian2modelfitting.simulator.**set_parameter_value**(*identifier*, *value*)
change parameter value in cpp file

brian2modelfitting.simulator.**set_states**(*init_dict*, *values*)
set parameters values in the file for the NeuronGroup

brian2modelfitting.utils module

class brian2modelfitting.utils.**ProgressBar**(*total=None*, ***kwds*)
Bases: object

Setup for tqdm progress bar in Fitter

brian2modelfitting.utils.**callback_none**(*params*, *errors*, *best_params*, *best_error*, *index*)
Non-verbose callback

brian2modelfitting.utils.**callback_setup**(*set_type*, *n_rounds*)
Helper function for callback setup in Fitter, loads option: 'text', 'progressbar' or custion FunctionType

brian2modelfitting.utils.**callback_text**(*params*, *errors*, *best_params*, *best_error*, *index*)
Default callback print-out for Fitters

brian2modelfitting.utils.**make_dic**(*names*, *values*)
Create dictionary based on list of strings and 2D array

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

G

I

M

N

O

P

R

S

T