brian2modelfitting

Release 0.4+git

Jun 21, 2023

Contents

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

Please report bugs at the GitHub issue tracker or at the Brian 2 discussion forum. The latter is also a place to discuss feature requests or potential contributions.

CHAPTER 1

Model fitting

This toolbox allows the user to find the best fit of the unknown free parameters for recorded traces and spike trains. It also supports simulation-based inference, where instead of point-estimated parameter values, a full posterior distribution over the parameters is computed.

By default, the toolbox supports a range of global derivative-free optimization methods, that include popular methods for model fitting: differential evolution, particle swarm optimization and covariance matrix adaptation (provided by the `nevergrad`, a gradient-free optimization platform) as well as Bayesian optimization for black box functions (provided by `scikit-optimize`, a sequential model-based optimization library). On the other hand, simulation-based inference is the process of finding parameters of a simulator from observations by taking a Bayesian approach, in our case, via sequential neural posterior estimation, likelihood estimation or ratio estimation (provided by the `sbi`), where neural densitiy estimator, a deep neural network allowing probabilistic association between the data and underlying parameter space, is trained. After the network is trained, the approximated posterior distribution is available.

Just like Brian 2 simulator itself, the *`brian2modelfitting`* toolbox is designed to be easy to use and to 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*

and a simulation-based inference class:

- *Inferencer*

All classes expect a model and the data as an input and return either the best fit of each parameter with the corresponding error, or a posterior distribution over unknown parameters. The toolbox can optimize over multiple traces (e.g. input currents) at the same time. It also allows the possiblity of simultaneous fitting/inferencing by taking into account multiple output variables including spike trains.

In following documentation we assume that *brian2modelfitting* has been installed and imported as follows:

```
from brian2modelfitting import *
```

Installation

To install the toolbox alongside Brian 2 simulator, use `pip` as follows:

```
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 following command inside the *`brian2modelfitting`* root directory:

```
pytest
```

2.1.2 How it works

Fitting

Model fitting script requires three components:

- a **fitter**: object that will perform the optimization
- a **metric**: objective function
- an **optimizer**: optimization algorithm

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 the objective function can be easily swapped out by a user-defined custom implementation.

`Fitter` objects require a 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` - the time step
- `input` - a dictionary with the name of the input variable and a set of

input traces (list or array) - `output` - a dictionary with the name of the output variable(s) and a set of goal output (traces/spike trains) (list or array) - `n_samples` - a 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` - a number of rounds to optimize over
- parameters with ranges to be optimized over

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

```
param_name = [lower_bound, upper_bound]
```

Ready to use elements

Optimization classes:

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

Optimization algorithms:

- *NevergradOptimizer*
- *SkoptOptimizer*

Metrics:

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

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

```
fitter = TraceFitter(model=model,
                     input={'I': inp_traces},
                     output={'v': out_traces},
                     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 round, 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*.

Simulation-based inference

The *Inferencer* class has to be initialized within the script that will perform a simulation-based inference procedure.

Initialization of *Inferencer* requires:

- `dt` - the time step in Brian 2 units.
- `model` - single cell model equations, defined as either string or as `brian2.Equation` object.

- `input` - a dictionary where key corresponds to the name of the input variable as defined in `model` and value corresponds to an array of input traces.
- `output` - a dictionary where key corresponds to the name of the output variable as defined in `model` and value corresponds to an array of recorded traces and/or spike trains.

```
inferencer = Inferencer(dt=0.1*ms, model=eqs,
                        input={'I': inp_traces*amp},
                        output={'v': out_traces*mV})
```

Optionally, arguments to be passed to the constructor are:

- `features` - a dictionary of callables that take the voltage trace and/or spike trains and output summary statistics. Keys correspond to output variable names, while values are lists of callables. If features are not provided, automatic feature extraction will be performed either by using the default multi-layer perceptron or by using the user-provided embedding network.
- `method` - a string that defines an integration method.
- `threshold` - optional string that defines the condition which produces spikes. It should be a single line boolean expression.
- `reset` - an optional (multi-line) string that that holds the code to execute on reset.
- `refractory` - can be either Boolean expression or string. Defines 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'`.
- `param_init` - a dictionary of state variables to be initialized with respective values, i.e., initial conditions.

```
inferencer = Inferencer(dt=dt, model=eqs_inf,
                        input={'I': inp_trace*amp},
                        output={'v': out_traces*mV},
                        features={'v': voltage_feature_list},
                        method='exponential_euler',
                        threshold='v > -50*mV',
                        reset='v = -70*mV',
                        param_init={'v': -70*mV})
```

Inference

After the *Inferencer* class is instantiated, the simplest and the most convenient way to start with the inferencer procedure is by calling *infer* method on *Inferencer* object.

In the nutshell, infer method returns the trained neural posterior object, which may or may not be used by the user, but it has to exist. There are two possible approaches:

- amortized inference
- multi-round inference

If the number of inference rounds is 1, then amortized inference will be performed. Otherwise if the number of inference rounds is 2 or above, the focused multi-round inference will be performed. Multi-round inference, unlike the amortized one, is focused on a particular observation, where in each new round of inference, samples are drawn from the posterior distribution conditioned exactly by this observation. This process can be repeated aribtrarily many times to get increasingly better approximations of the the posterior distribution.

The infer method requires:

- `n_samples` - the number of samples from which the neural posterior will be learnt.

or:

- `theta` - sampled prior.
- and `x` - summary statistics.

along with the:

- `params` - a dictionary of bounds for each free parameter defined in the `model`. Keys should correspond to names of parameters as defined in the model equations, values are lists with lower and upper bounds with quantities of respective parameter.

The simplest way to start the inference process is by calling:

```
posterior = inferencer.infer(n_samples=1000,
                             gl=[10*nS, 100*nS],
                             C=[0.1*nF, 10*nF])
```

Optionally, user can defined the following arguments:

- `n_rounds` - if it is set to 1, amortized inference will be performed. Otherwise, if `n_rounds` is integer larger than 1, multi-round inference will be performed. This is only valid if the posterior has not yet been defined. Otherwise, if this method is called after the posterior has already been built, multi-round inference is performed, e.g. repeated calling of `~brian2modelfitting.inferencer.Inferencer.infer` method or manually building the posterior by approaching the inference with flexible inference.
- `inference_method` - either SNPE, SNLE or SNRE.
- `density_estimator_model` - string that defines the type of density estimator to be created. Either `mdn`, `made`, `maf`, `nsf` for SNPE and SNLE, or `linear`, `mlp`, `resnet` for SNRE.
- `inference_kwargs` - a dictionary that holds additional keyword arguments for the *`init_inference`*.
- `train_kwargs` - a dictionary that holds additional keyword arguments for *`train`*.
- `posterior_kwargs` - a dictionary that holds additional keyword arguments for *`build_posterior`*.
- `restart` - when the method is called for a second time, set to True if amortized inference should be performed. If False, multi-round inference with the existing posterior will be performed.
- `sbi_device` a string that defines the device on which the `sbi` and subseqently the `torch` will operate. By default this is set to `cpu` and it is advisable to remain so for most cases. In cases where the user provides custom embedding network through `inference_kwargs` argument, which will be trained more efficiently by using GPU, device should be set accordingly to `gpu`.

A bit more comprehensive specification of the infer call is showcased below:

```
posterior = inferencer.infer(n_samples=5_000,
                             n_rounds=3,
                             inference_method='SNPE',
                             density_estimator_model='mdn',
                             restart=True,
                             sbi_device='cpu',
                             gl=[10*nS, 100*nS],
                             C=[0.1*nF, 10*nF])
```

Remarks

For a better understanding, please go through examples that go step-by-step through the entire process. Currently, there are two tutorials: the one that is covering simple interface, appropriate for the regular user, and the one that goes a bit more in-depth by using flexible interface, and shows how to manually go through the process of inference, storing/loading the training data and the trained neural density estimator, parameter space visualization, conditioning, etc.

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.

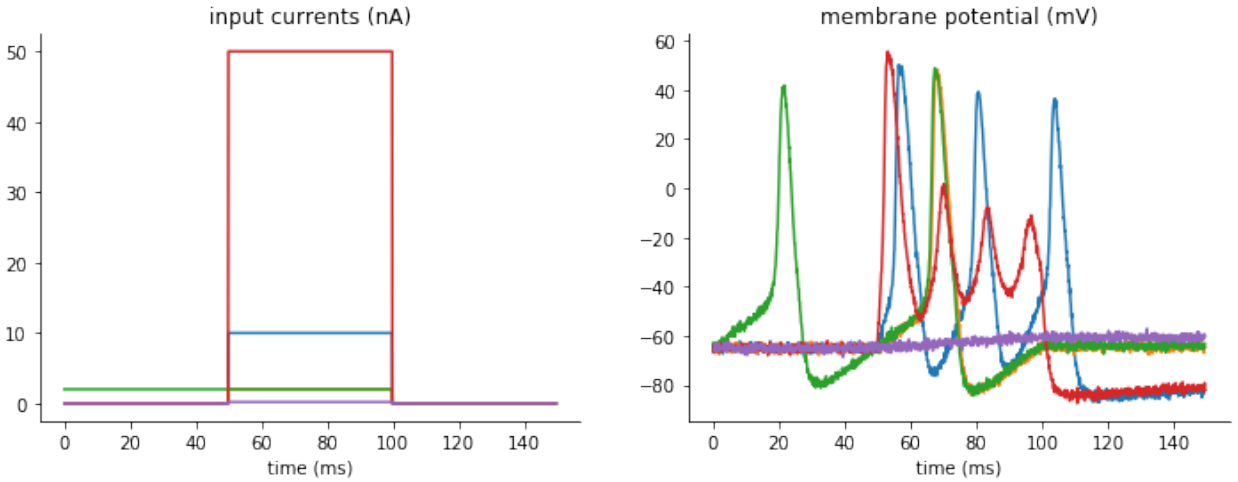
We start by importing `brian2` and `brian2modelfitting`:

```
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)/
  (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={'I': inp_trace*amp},
                     output={'v': out_trace*mV},
```

(continues on next page)

(continued from previous page)

```
                        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:

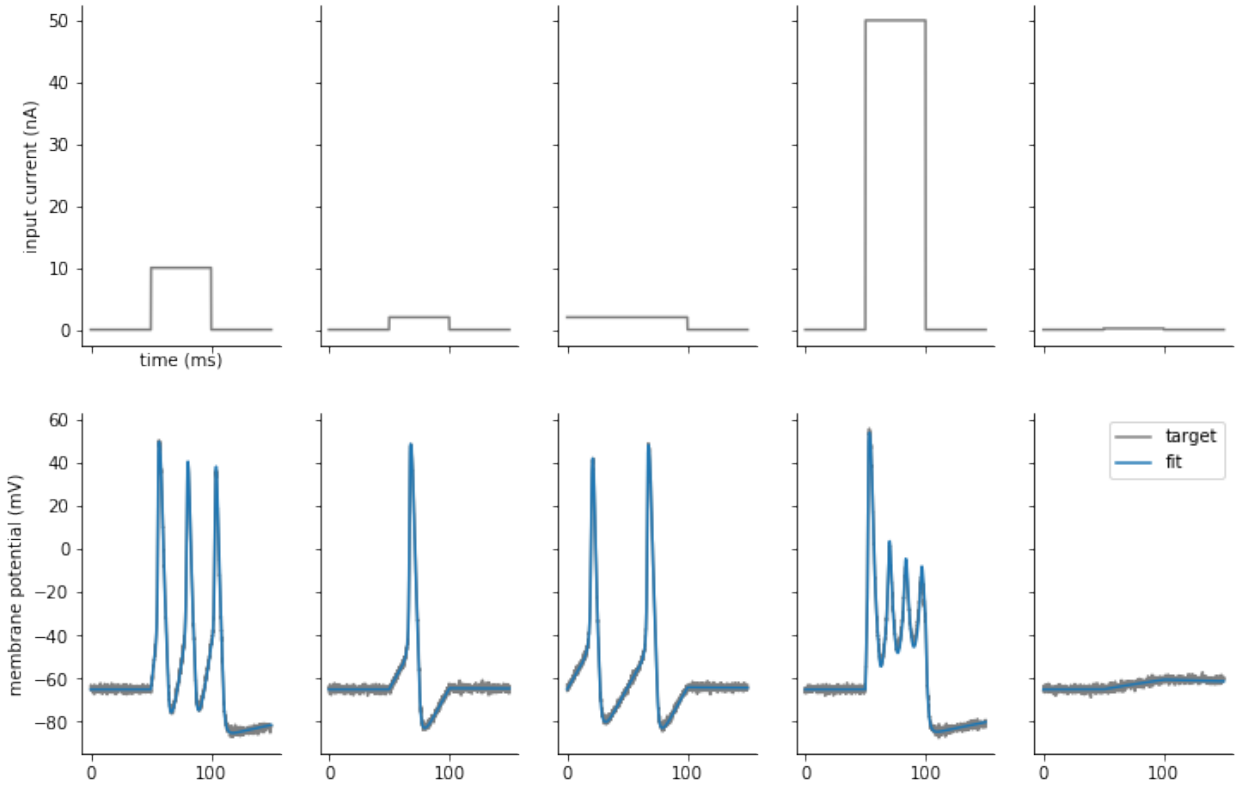
```
Round 0: fit [9.850944960633812e-05, 5.136956717618642e-05, 1.132001753695881e-07]␣
↪with error: 0.00023112503428419085
Round 1: fit [2.5885625978001192e-05, 5.994175009416155e-05, 1.132001753695881e-07]␣
↪with error: 0.0001351283127819249
Round 2: fit [2.358033085911261e-05, 5.286319601683492e-05, 7.255743458079185e-08]␣
↪with error: 8.600916130059129e-05
Round 3: fit [2.013515980650059e-05, 4.5888592959196316e-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

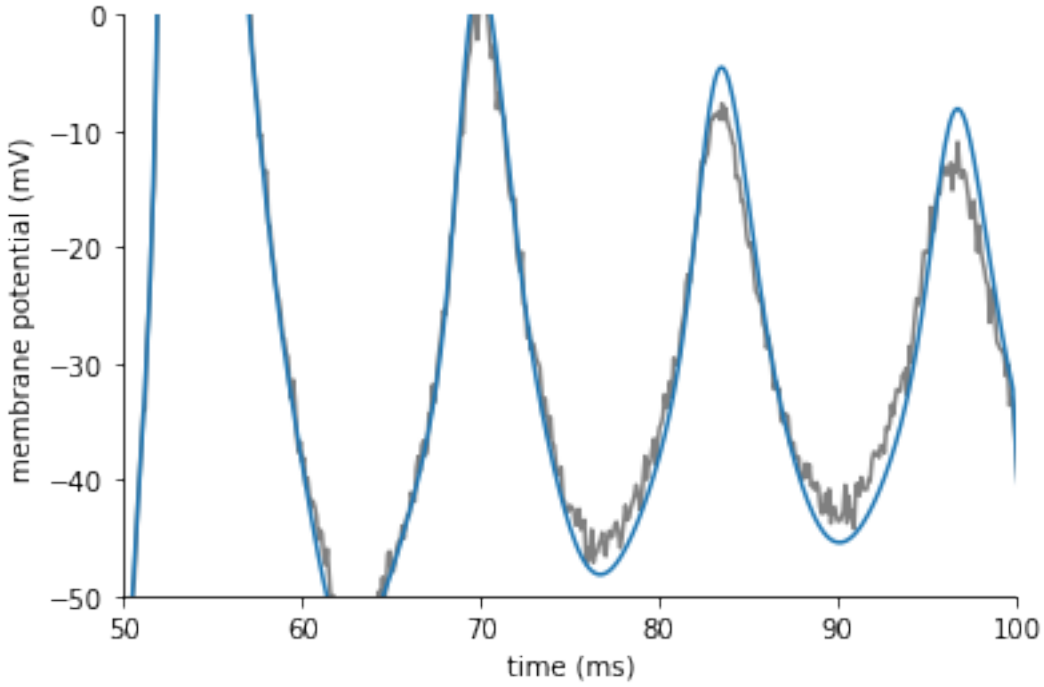
To generate the traces that correspond to the new best fit parameters of the model, you can use the *generate_traces* method.

```
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

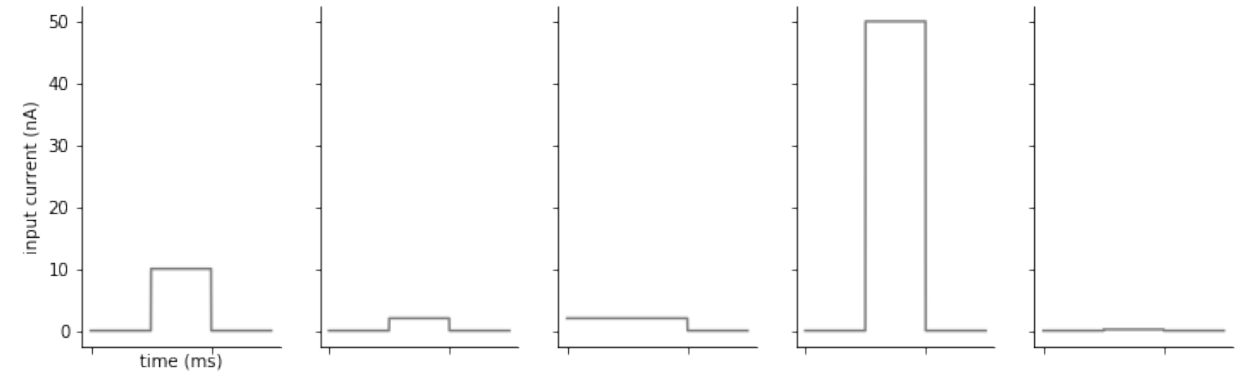
When using *TraceFitter*, you can further refine the fit by applying a standard least squares fitting algorithm (e.g. Levenberg–Marquardt), by calling *refine*. By default, this will start from the previously found best parameters:

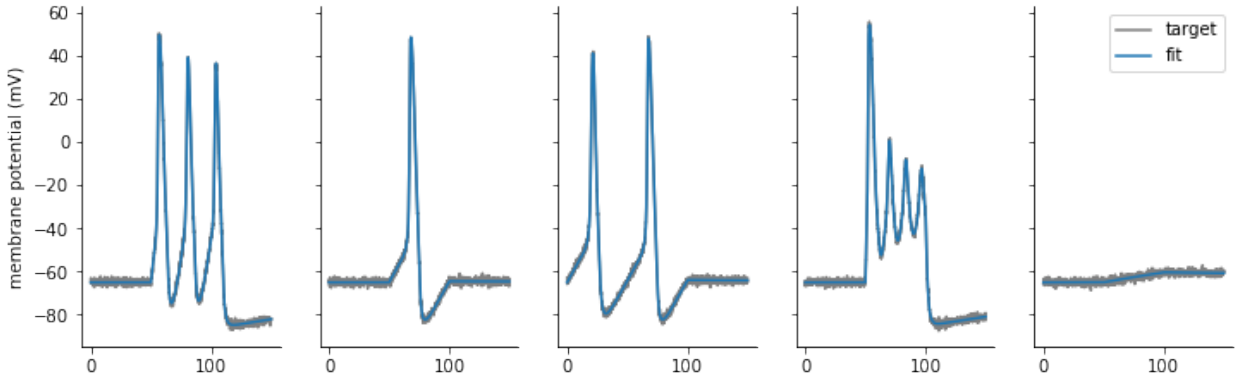
```
refined_params, result_info = fitter.refine()
```

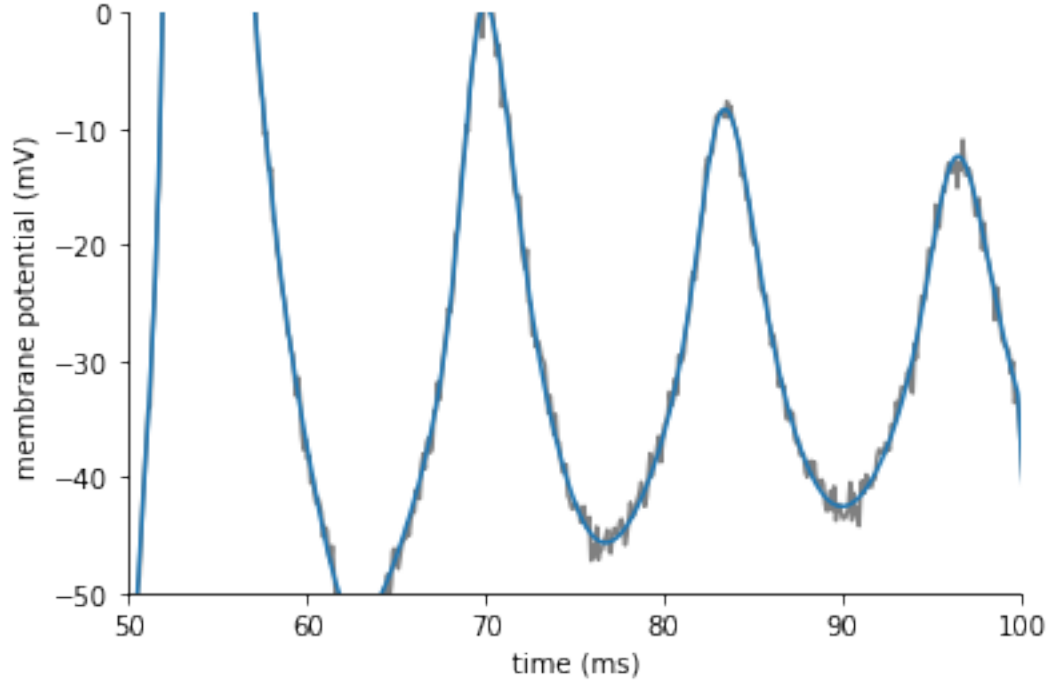
We can now generate traces with the refined parameters:

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

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







2.1.4 Tutorial: Inferencer

In this tutorial, we use simulation-based inference on the Hodgkin-Huxley neuron model, [Hodgkin1952], where different scenarios are considered. The data are synthetically generated from the model which enables the comparisson

of the optimized parameter values with the ground truth parameter values as used in the generative model.

We start with importing basic packages and modules. Note that `tutorial_sbi_helpers` contains three different functions that are used extensively throughout this tutorial. Namely, `plot_traces` is used for visualization of the input current and output voltage traces, and optionally for visualization of spike trains or sampled traces obtained by using an estimated posterior. In order to detect and capture spike events in a given voltage trace, we use `spike_times` function. Finally, `plot_cond_coeff_mat` is used to visualize conditional coerrelation matrix. For a detailed overview of the mechanism of each of the functions, download the script: `tutorial_sbi_helpers.py`

```
from brian2 import *
from brian2modelfitting import Inferencer
from scipy.stats import kurtosis as kurt
from tutorial_sbi_helpers import (spike_times,
                                  plot_traces,
                                  plot_cond_coeff_mat)
```

Now, let's load the input current and output voltage traces by using `NumPy`. Note that all functions available in `NumPy`, as well as in `Matplotlib`, are implicitly available after `brian2` was imported.

```
inp_traces = load('input_traces_synthetic.npy').reshape(1, -1)
out_traces = load('output_traces_synthetic.npy').reshape(1, -1)
```

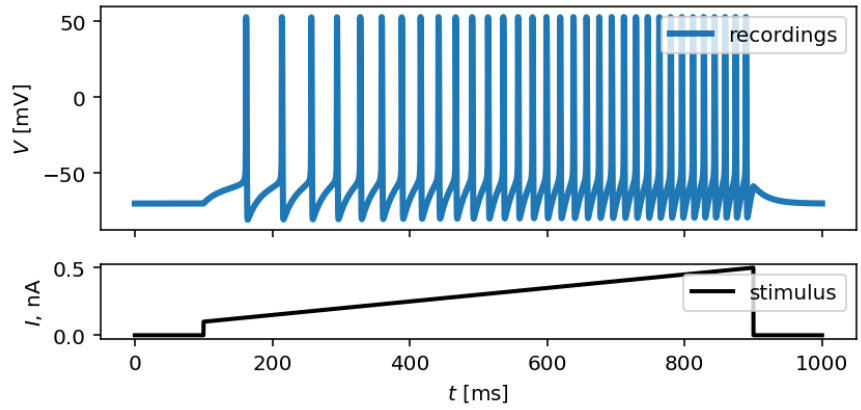
The data is generated by running `generate_traces_synthetic.py` script: `generate_traces_synthetic.py`

By setting the time step, we are able to set up the time domain. From the array of time steps, we also define the exact time when the stimulus starts and when it ends by computing `stim_start` and `stim_end`, respectively. This will come handy later during the feature extraction process.

```
dt = 0.05*ms
t = arange(0, inp_traces.size*dt/ms, dt/ms)
stim_start, stim_end = t[where(inp_traces[0, :] != 0)[0][[0, -1]]]
```

By calling `plot_traces` and passing the array of time steps, the input current and output voltage traces, we obtain the visualization of a synthetic neural activity recordings:

```
fig, ax = plot_traces(t, inp_traces, out_traces)
```



Toy-example: infer two free parameters

The first scenario we cover is a simple inference procedure of two unknown parameters in the Hodgkin-Huxley neuron model. The parameters to infer are the maximal value of sodium, $\overline{g}_{\mathrm{Na}}$, and potassium electrical conductances, $\overline{g}_{\mathrm{K}}$.

By following the standard practice from Brian 2 simulator, we have to define parameters of the model, initial conditions for differential equations that describe the model, and the model itself:

```
# set parameters of the model
E_Na = 53*mV
E_K = -107*mV
E_l = -70*mV
VT = -60.0*mV
g_l = 10*nS
Cm = 200*pF

# set ground truth parameters, which are unknown from the model's perspective
ground_truth_params = {'g_Na': 32*uS,
                       'g_K': 1*uS}

# define initial conditions
init_conds = {'v': 'E_l',
              'm': '1 / (1 + beta_m / alpha_m)',
              'h': '1 / (1 + beta_h / alpha_h)',
              'n': '1 / (1 + beta_n / alpha_n)'}

# define the Hodgkin-Huxley neuron model
eqs = '''
    # non-linear set of ordinary differential equations
    dv/dt = - (g_Na * m ** 3 * h * (v - E_Na)
               + g_K * n ** 4 * (v - E_K)
               + g_l * (v - E_l) - I) / Cm : volt
    dm/dt = alpha_m * (1 - m) - beta_m * m : 1
    dn/dt = alpha_n * (1 - n) - beta_n * n : 1
    dh/dt = alpha_h * (1 - h) - beta_h * h : 1

    # time independent rate constants for a channel activation/inactivation
    alpha_m = ((-0.32 / mV) * (v - VT - 13.*mV))
              / (exp((-(v - VT - 13.*mV)) / (4.*mV)) - 1) / ms : Hz
    beta_m = ((0.28/mV) * (v - VT - 40.*mV))
             / (exp((v - VT - 40.*mV) / (5.*mV)) - 1) / ms : Hz
    alpha_h = 0.128 * exp(-(v - VT - 17.*mV) / (18.*mV)) / ms : Hz
    beta_h = 4 / (1 + exp((-(v - VT - 40.*mV)) / (5.*mV))) / ms : Hz
    alpha_n = ((-0.032/mV) * (v - VT - 15.*mV))
              / (exp((-(v - VT - 15.*mV)) / (5.*mV)) - 1) / ms : Hz
    beta_n = 0.5 * exp(-(v - VT - 10.*mV) / (40.*mV)) / ms : Hz

    # free parameters
    g_Na : siemens (constant)
    g_K : siemens (constant)
    '''
```

Since the output of the model is extremely high-dimensional, and since we are interested only in a few hand-picked features that will capture the gist of the neuronal activity, we start the inference process by defining a list of summary functions.

Each summary feature is obtained by calling a single-valued function on each trace generated by using a sampled prior distribution over unknown parameters. In this case, we consider the maximal value, mean and standard deviatian of action potential, and the membrane resting potential.

```
v_features = [
    # max action potential
    lambda x: np.max(x[(t > stim_start) & (t < stim_end)]),
```

(continues on next page)

(continued from previous page)

```
    # mean action potential
    lambda x: np.mean(x[(t > stim_start) & (t < stim_end)]),
    # std of action potential
    lambda x: np.std(x[(t > stim_start) & (t < stim_end)]),
    # membrane resting potential
    lambda x: np.mean(x[(t > 0.1 * stim_start) & (t < 0.9 * stim_start)])
]
```

Inferencer

The minimum set of arguments for the *Inferencer* class constructor are the time step, `dt`, input data traces, `input`, output data traces, `output`, and the model that will be used for the inference process, `model`. Input and output traces should have the number of rows that corresponds to the number of observed traces, and the number of columns should be equal to the number of time steps in each trace.

Here, we define additional arguments such as: `method` to define an integration technique used for solving the set of differential equations, `threshold` to define a condition that produce a single spike, `refractory` to define a condition under which a neuron remains refractory, and `param_init` to define a set of initial conditions. We also define the set of summary features that is used to represent the data instead of using the entire trace. Summary features are passed to the inference algorithm via `features` argument.

```
inferencer = Inferencer(dt=dt, model=eqs,
                        input={'I': inp_traces*amp},
                        output={'v': out_traces*mV},
                        features={'v': v_features},
                        method='exponential_euler',
                        threshold='m > 0.5',
                        refractory='m > 0.5',
                        param_init=init_conds)
```

After the `inferencer` is instantiated, we may begin the inference process by calling *infer* and defining the total number of samples that are used for the training of a neural density estimator. We use the sequential neural posterior estimation algorithm (SNPE), proposed in [Greenberg2019].

Posterior

Neural density estimator learns the probabilistic mapping of the input data, i.e., sampled parameter values given a prior distribution, and the output data, i.e., summary features extracted from the traces, obtained by solving the model with the corresponding set of sampled parameters from the input data.

We can choose the inference method and the estimator model , but only arguments that *infer* requires are the number of samples (in case of running the inference process for the first time), `n_samples`, and upper and lower bounds for each unknown parameter.

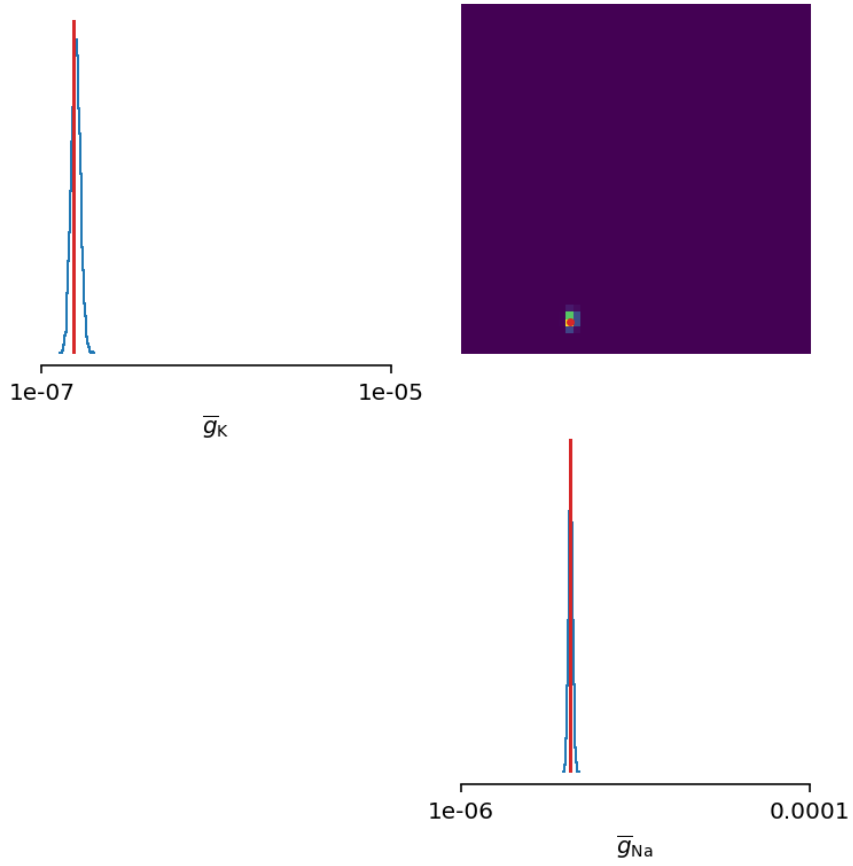
```
posterior = inferencer.infer(n_samples=15_000,
                             n_rounds=1,
                             inference_method='SNPE',
                             density_estimator_model='maf',
                             g_Na=[1*uS, 100*uS],
                             g_K=[0.1*uS, 10*uS])
```

After inference is completed, the estimated posterior distribution can be analyzed by observing the pairwise relationship between each pair of parameters. But before, we have to draw samples from the estimated posterior as follows:

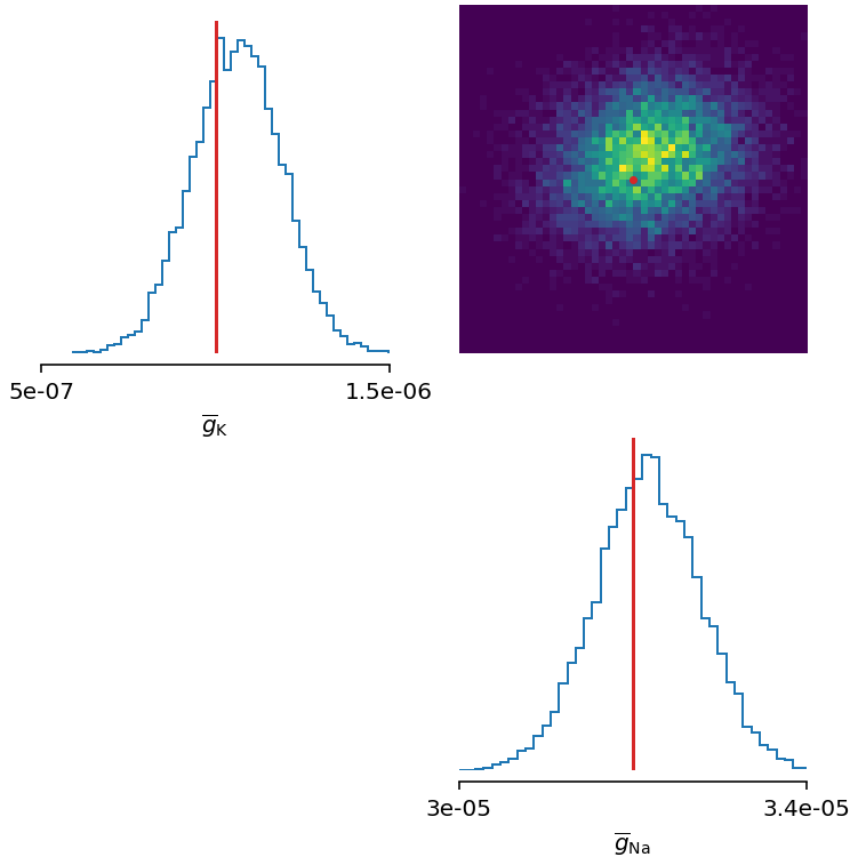
```
samples = inferencer.sample((10_000, ))
```

The samples are stored inside the *Inferencer* object and are available through `samples` variable. We create a visual representation of the pairwise relationship of the posterior as follows:

```
limits = {'g_Na': [1*uS, 100*uS],
          'g_K': [0.1*uS, 10*uS]}
labels = {'g_Na': r'$\overline{g}_{Na}$',
          'g_K': r'$\overline{g}_{K}$'}
fig, ax = inferencer.pairplot(limits=limits,
                              labels=labels,
                              ticks=limits,
                              points=ground_truth_params,
                              points_offdiag={'markersize': 5},
                              points_colors=['C3'],
                              figsize=(6, 6))
```



Let's zoom in a bit:



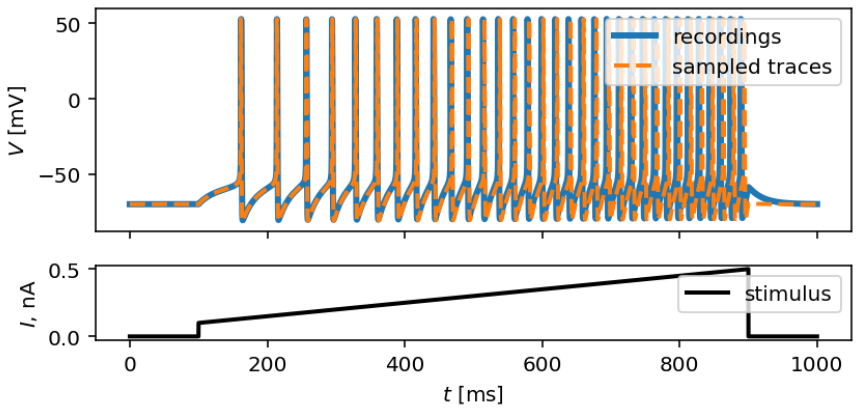
The inferred posterior is plotted against the ground truth parameters, and as can be seen, the ground truth parameters are located in high-probability regions of the estimated distribution.

To further substantiate this, let's now see the traces simulated from a single set of parameters sampled from the posterior:

```
inf_traces = inferencer.generate_traces()
```

We again use the `plot_traces` function as follows:

```
fig, ax = plot_traces(t, inp_traces, out_traces, inf_traces=array(inf_traces/mV))
```



Additional free parameters

The simple scenarios where only 2 parameters are considered works quite well using synthetic data traces. What if we have a larger number of unkown parameters? Let's now consider additional unkown parameters for the same model as before. In addition to the unknown maximal values of the electrical conductance of the sodium and potassium channels, the membrane capacity, C_m, and the maximal value of the electrical conductance of the leakage ion channel, $\overline{g}_l$, are also unknown.

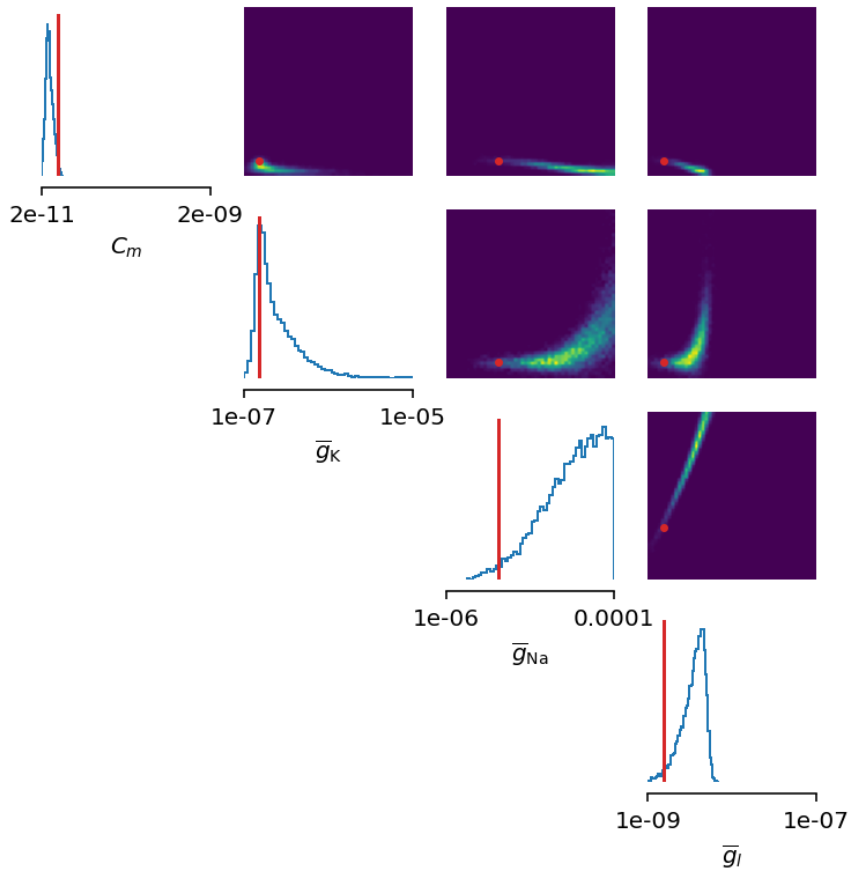
We can try to do the same as before with a bit more training data:

```
del Cm, g_l

# set parameters, initial condition and the model
E_Na = 53*mV
E_K = -107*mV
E_l = -70*mV
VT = -60.0*mV

ground_truth_params = {'g_Na': 32*uS,
                       'g_K': 1*uS,
                       'g_l': 10*nS,
                       'Cm': 200*pF}

init_conds = {'v': 'E_l',
              'm': '1 / (1 + beta_m / alpha_m)',
              'h': '1 / (1 + beta_h / alpha_h)',
              'n': '1 / (1 + beta_n / alpha_n)'}

eqs = '''
    # non-linear set of ordinary differential equations
    dv/dt = - (g_Na * m ** 3 * h * (v - E_Na)
               + g_K * n ** 4 * (v - E_K)
               + g_l * (v - E_l) - I) / Cm : volt
    dm/dt = alpha_m * (1 - m) - beta_m * m : 1
    dn/dt = alpha_n * (1 - n) - beta_n * n : 1
    dh/dt = alpha_h * (1 - h) - beta_h * h : 1

    # time independent rate constants for activation and inactivation
    alpha_m = ((-0.32 / mV) * (v - VT - 13.*mV))
              / (exp((-(v - VT - 13.*mV)) / (4.*mV)) - 1) / ms : Hz
    beta_m = ((0.28/mV) * (v - VT - 40.*mV))
             / (exp((v - VT - 40.*mV) / (5.*mV)) - 1) / ms : Hz
    alpha_h = 0.128 * exp(-(v - VT - 17.*mV) / (18.*mV)) / ms : Hz
    beta_h = 4 / (1 + exp((-(v - VT - 40.*mV)) / (5.*mV))) / ms : Hz
    alpha_n = ((-0.032/mV) * (v - VT - 15.*mV))
              / (exp((-(v - VT - 15.*mV)) / (5.*mV)) - 1) / ms : Hz
    beta_n = 0.5 * exp(-(v - VT - 10.*mV) / (40.*mV)) / ms : Hz

    # free parameters
    g_Na : siemens (constant)
    g_K : siemens (constant)
    g_l : siemens (constant)
    Cm : farad (constant)
    '''

# infer the posterior using the same configuration as before
inferencer = Inferencer(dt=dt, model=eqs,
                        input={'I': inp_traces*amp},
```

(continues on next page)

(continued from previous page)

```
                          output={'v': out_traces*mV},
                          features={'v': v_features},
                          method='exponential_euler',
                          threshold='m > 0.5',
                          refractory='m > 0.5',
                          param_init=init_conds)

posterior = inferencer.infer(n_samples=20_000,
                             n_rounds=1,
                             inference_method='SNPE',
                             density_estimator_model='maf',
                             g_Na=[1*uS, 100*uS],
                             g_K=[0.1*uS, 10*uS],
                             g_l=[1*nS, 100*nS],
                             Cm=[20*pF, 2*nF])

# finally, sample and visualize the posterior distribution
samples = inferencer.sample((10_000, ))

limits = {'g_Na': [1*uS, 100*uS],
          'g_K': [0.1*uS, 10*uS],
          'g_l': [1*nS, 100*nS],
          'Cm': [20*pF, 2*nF]}
labels = {'g_Na': r'$\overline{g}_{Na}$',
          'g_K': r'$\overline{g}_{K}$',
          'g_l': r'$\overline{g}_{l}$',
          'Cm': r'$C_{m}$'}
fig, ax = inferencer.pairplot(limits=limits,
                              labels=labels,
                              ticks=limits,
                              points=ground_truth_params,
                              points_offdiag={'markersize': 5},
                              points_colors=['C3'],
                              figsize=(6, 6))
```



This could have been expected. The posterior distribution is estimated poorly using a simple approach as in the toy example.

Yes, we can play around with hyper-parameters and tuning the neural density estimator, but with this apporach we will not get far.

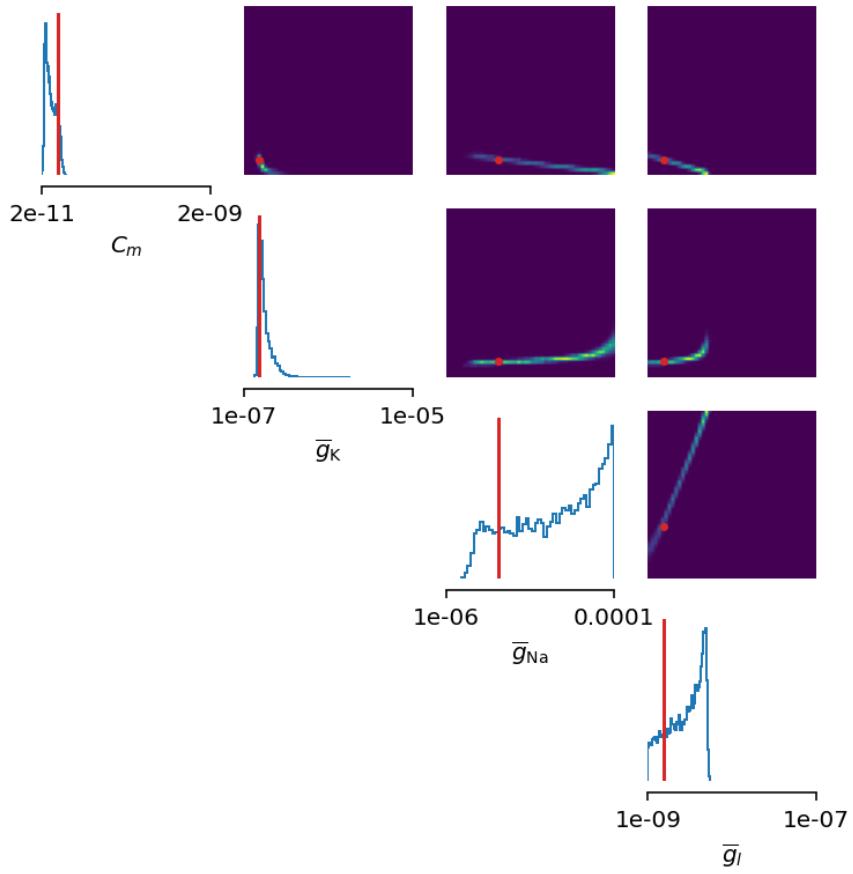
We can, however, try with the non-amortized (or focused) approach, meaning we perform multi-round inference, where each following round will use the posterior from the previous one to sample new input data for the training, rather than using the same prior distribution as defined in the beginning. This approach yields additional advantage - the number of samples may be considerably lower, but it will lead to the posterior that is no longer being amortized - it is accurate only for a specific observation.

```
# note that the only difference is the number of rounds of inference
posterior = inferencer.infer(n_samples=10_000,
                             n_rounds=2,
                             inference_method='SNPE',
                             density_estimator_model='maf',
                             restart=True,
                             g_Na=[1*uS, 100*uS],
                             g_K=[0.1*uS, 10*uS],
                             g_l=[1*nS, 100*nS],
                             Cm=[20*pF, 2*nF])

samples = inferencer.sample((10_000, ))

fig, ax = inferencer.pairplot(limits=limits,
                              labels=labels,
```

(continues on next page)

(continued from previous page)

```
                                ticks=limits,
                                points=ground_truth_params,
                                points_offdiag={'markersize': 5},
                                points_colors=['C3'],
                                figsize=(6, 6))
```



This seems as a promising approach for parameters that already have the high-probability regions of the posterior distribution around ground-truth values. For other parameters, this leads to further deterioration of posterior estimates.

So, we may wonder, how else can we improve the neural density estimator accuracy?

Currently, we use only four features to describe extremely complex output of a neural model and should probably create a more comprehensive and more descriptive set of summary features. If we want to include data related to spikes in summary statistics, it is necessary to perform multi-objective optimization since we will observe spike trains as an output in addition to voltage traces.

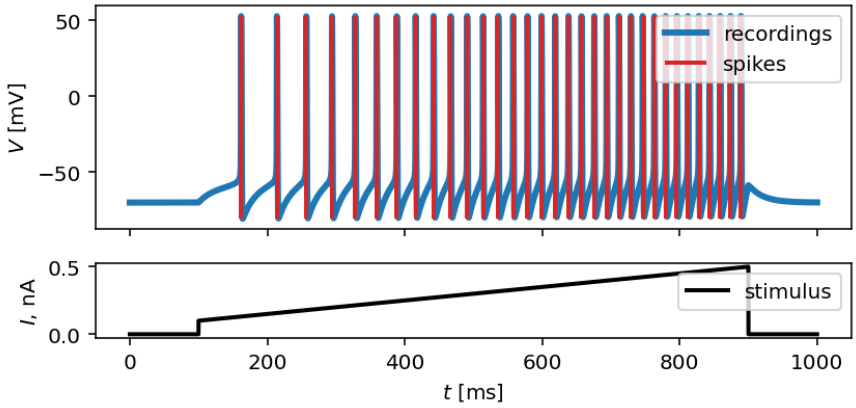
Multi-objective optimization

In order to use spikes, we have to have some observation to pass to the *Inferencer*. We can utilize `spike_times` as follows:

```
spike_times_list = [spike_times(t, out_trace) for out_trace in out_traces]
```

To visually prove that spike times are indeed correct, we use `plot_traces`:

```
fig, ax = plot_traces(t, inp_traces, out_traces, spike_times_list[0])
```



Now, let's create additional features that will be applied to voltage traces, and a few features that will be applied to spike trains:

```
def voltage_deflection(x):
    voltage_base = np.mean(x[t < stim_start])
    stim_end_idx = np.where(t >= stim_end)[0][0]
    steady_state_voltage_stimend = np.mean(x[stim_end_idx-10:stim_end_idx-5])
    return steady_state_voltage_stimend - voltage_base


v_features = [
    # max action potential
    lambda x: np.max(x[(t > stim_start) & (t < stim_end)]),
    # mean action potential
    lambda x: np.mean(x[(t > stim_start) & (t < stim_end)]),
    # std of action potential
    lambda x: np.std(x[(t > stim_start) & (t < stim_end)]),
    # kurtosis of action potential
    lambda x: kurt(x[(t > stim_start) & (t < stim_end)], fisher=False),
    # membrane resting potential
    lambda x: np.mean(x[(t > 0.1 * stim_start) & (t < 0.9 * stim_start)]),
    # the voltage deflection between base and steady-state voltage
    voltage_deflection,
]

s_features = [
    # number of spikes in a train
    lambda x: x.size,
    # mean inter-spike interval
    lambda x: 0. if np.diff(x).size == 0 else np.mean(diff(x)),
    # time to first spike
    lambda x: 0. if x.size == 0 else x[0]
]
```

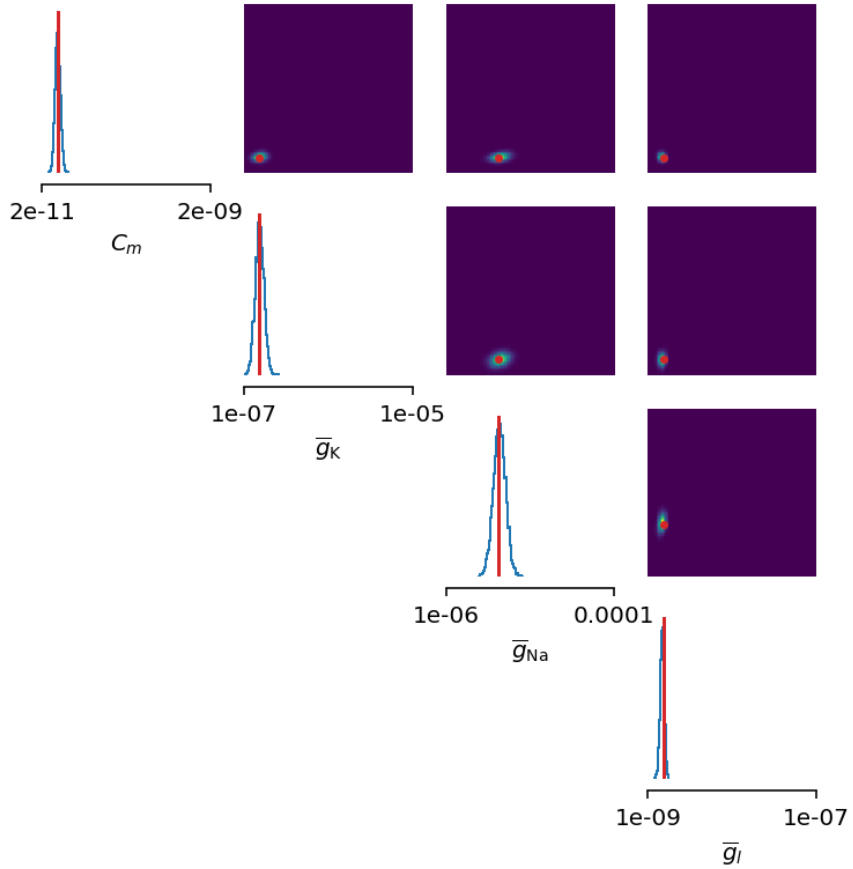
The rest of the inference process stays pretty much the same as before:

```
inferencer = Inferencer(dt=dt, model=eqs,
                        input={'I': inp_traces*amp},
                        output={'v': out_traces*mV, 'spikes': spike_times_list},
                        features={'v': v_features, 'spikes': s_features},
```

(continues on next page)

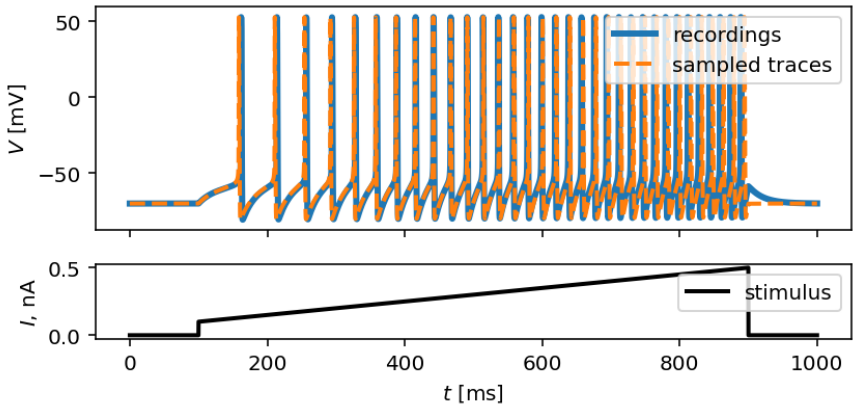
(continued from previous page)

```
                          method='exponential_euler',
                          threshold='m > 0.5',
                          refractory='m > 0.5',
                          param_init=init_conds)

posterior = inferencer.infer(n_samples=20_000,
                             n_rounds=1,
                             inference_method='SNPE',
                             density_estimator_model='maf',
                             g_Na=[1*uS, 100*uS],
                             g_K=[0.1*uS, 10*uS],
                             g_l=[1*nS, 100*nS],
                             Cm=[20*pF, 2*nF])

samples = inferencer.sample((10_000, ))

fig, ax = inferencer.pairplot(limits=limits,
                              labels=labels,
                              ticks=limits,
                              points=ground_truth_params,
                              points_offdiag={'markersize': 5},
                              points_colors=['C3'],
                              figsize=(6, 6))
```



Let's also visualize the sampled trace, this time using the mean of ten thousands drawn samples:

```
inf_traces = inferencer.generate_traces(n_samples=10_000, output_var='v')

fig, ax = plot_traces(t, inp_traces, out_traces, inf_traces=array(inf_traces/mV))
```



Okay, now we are clearly getting somewhere and this should be a strong indicatior of the importance of crafting quality summary statistics.

Still, the summary statistics can be a huge bottleneck and can set back the training of a neural density estimator. For this reason automatic feature extraction can be considered instead.

Automatic feature extraction

To enable automatic feature extraction, `features` argument simly should not be defined when instantiating an inferencer object. And that's it. Everything else happens behind the scenes without any need for additional user intervention. If the user wants to gain additional control over the extraction process, in addition to changing the hyperparameters, they can also define their own embedding neural network.

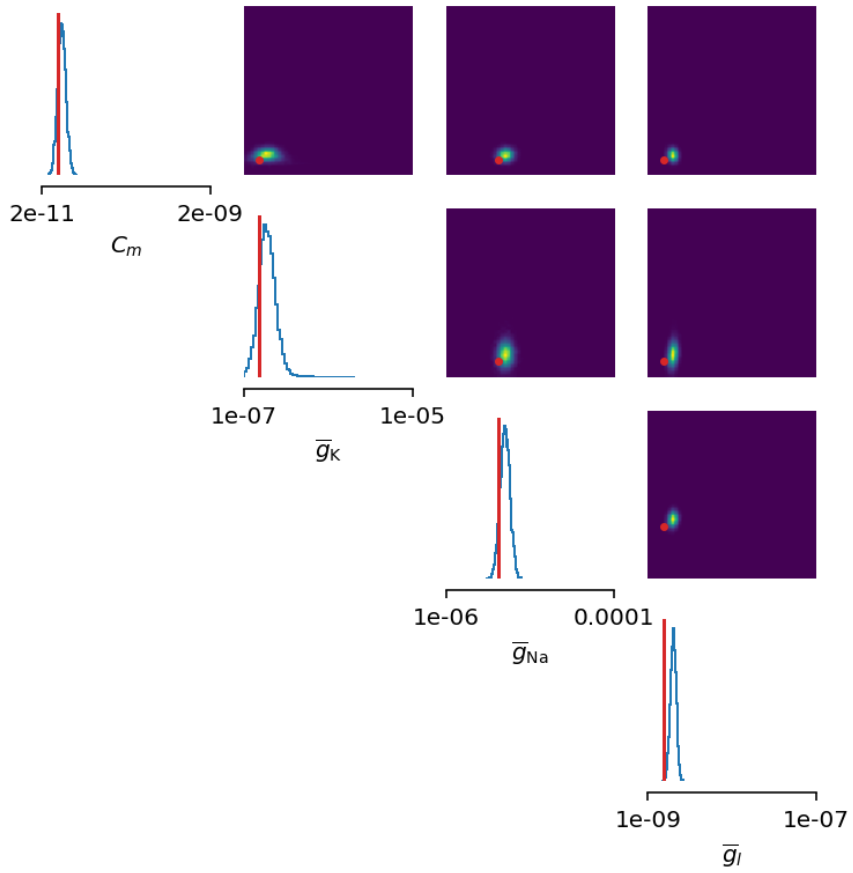
Default settings

```
inferencer = Inferencer(dt=dt, model=eqs,
                        input={'I': inp_traces*amp},
                        output={'v': out_traces*mV},
                        method='exponential_euler',
                        threshold='m > 0.5',
                        refractory='m > 0.5',
                        param_init=init_conds)

posterior = inferencer.infer(n_samples=20_000,
                             n_rounds=1,
                             inference_method='SNPE',
                             density_estimator_model='maf',
                             g_Na=[1*uS, 100*uS],
                             g_K=[0.1*uS, 10*uS],
                             g_l=[1*nS, 100*nS],
                             Cm=[20*pF, 2*nF])

samples = inferencer.sample((10_000, ))

fig, ax = inferencer.pairplot(limits=limits,
```

(continues on next page)

(continued from previous page)

```
                                labels=labels,
                                ticks=limits,
                                points=ground_truth_params,
                                points_offdiag={'markersize': 5},
                                points_colors=['C3'],
                                figsize=(6, 6))
```



Custom embedding network

Here, we demonstrate how to build a custom summary feature extractor and how to exploit the GPU processing power to speed up the inference process.

Note that the use of the GPU will result in the speed-up of computation time only if a custom automatic feature extractor uses techniques that are actually faster to compute on the GPU.

For this case, we use the YuleNet, a convolutional neural network, proposed in [Rodrigues2020]. The authors outline impresive results where the automatic feature extraction by using the YuleNet is capable of outperforming carefully hand-crafted features.

```
import torch
from torch import nn


class YuleNet(nn.Module):
```

(continues on next page)

(continued from previous page)

```
    """The summary feature extractor proposed in Rodrigues 2020.

    Parameters
    ----------
    in_features : int
        Number of input features should correspond to the size of a
        single output voltage trace.
    out_features : int
        Number of the features that are used for the inference process.

    Returns
    -------
    None

    References
    ----------
    * Rodrigues, P. L. C. and Gramfort, A. "Learning summary features
      of time series for likelihood free inference" 3rd Workshop on
      Machine Learning and the Physical Sciences (NeurIPS 2020). 2020.
    """
    def __init__(self, in_features, out_features):
        super().__init__()
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=8, kernel_size=64,
                               stride=1, padding=32, bias=True)
        self.relu1 = nn.ReLU()
        pooling1 = 16
        self.pool1 = nn.AvgPool1d(kernel_size=pooling1)

        self.conv2 = nn.Conv1d(in_channels=8, out_channels=8, kernel_size=64,
                               stride=1, padding=32, bias=True)
        self.relu2 = nn.ReLU()
        pooling2 = int((in_features // pooling1) // 16)
        self.pool2 = nn.AvgPool1d(kernel_size=pooling2)

        self.dropout = nn.Dropout(p=0.50)
        linear_in = 8 * in_features // (pooling1 * pooling2) - 1
        self.linear = nn.Linear(in_features=linear_in,
                                out_features=out_features)
        self.relu3 = nn.ReLU()

    def forward(self, x):
        if x.ndim == 1:
            x = x.view(1, 1, -1)
        else:
            x = x.view(len(x), 1, -1)
        x_conv1 = self.conv1(x)
        x_relu1 = self.relu1(x_conv1)
        x_pool1 = self.pool1(x_relu1)

        x_conv2 = self.conv2(x_pool1)
        x_relu2 = self.relu2(x_conv2)
        x_pool2 = self.pool2(x_relu2)

        x_flatten = x_pool2.view(len(x), 1, -1)
        x_dropout = self.dropout(x_flatten)

        x = self.relu3(self.linear(x_dropout))
```

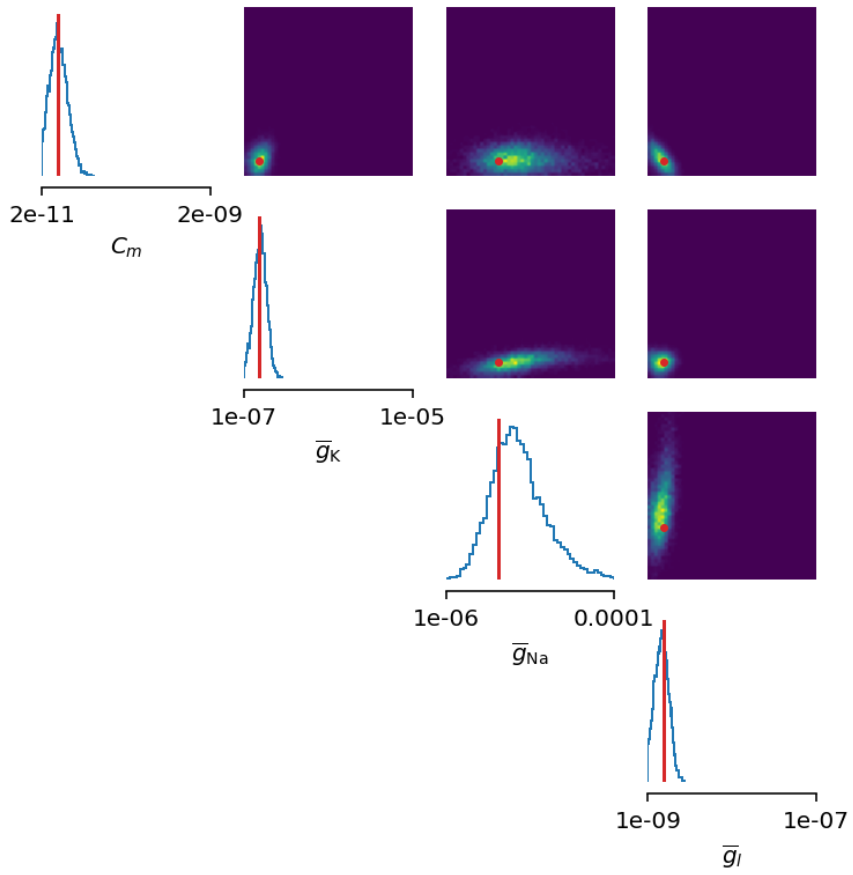
(continues on next page)

(continued from previous page)

```
        return x.view(len(x), -1)
```

In the following code, we also demonstrate how to control the hyperparameters of the density estimator by using additional keyword arguments in *infer* method:

```
in_features = out_traces.shape[1]
out_features = 10

inferencer = Inferencer(dt=dt, model=eqs,
                        input={'I': inp_traces*amp},
                        output={'v': out_traces*mV},
                        method='exponential_euler',
                        threshold='m > 0.5',
                        refractory='m > 0.5',
                        param_init=init_conds)

posterior = inferencer.infer(n_samples=20_000,
                             n_rounds=1,
                             inference_method='SNPE',
                             density_estimator_model='maf',
                             inference_kwargs={'embedding_net': YuleNet(in_features,␣
↪out_features)},
                             train_kwargs={'num_atoms': 10,
                                           'training_batch_size': 100,
                                           'use_combined_loss': True,
                                           'discard_prior_samples': True},
                             sbi_device='gpu',
                             g_Na=[1*uS, 100*uS],
                             g_K=[0.1*uS, 10*uS],
                             g_l=[1*nS, 100*nS],
                             Cm=[20*pF, 2*nF])

samples = inferencer.sample((10_000, ))

fig, ax = inferencer.pairplot(limits=limits,
                              labels=labels,
                              ticks=limits,
                              points=ground_truth_params,
                              points_offdiag={'markersize': 5},
                              points_colors=['C3'],
                              figsize=(6, 6))
```



Next steps

To learn more read the reference API and check out more examples available *here*.

References

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.

- *Nevergrad*
- *Scikit-Optimize (skopt)*
- *Custom Optimizer*

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.

- *Mean Square Error*
- *GammaFactor*
- *FeatureMetric*
- *Custom 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 $2\frac{|r_{\text{data}}-r_{\text{model}}|}{r_{\text{data}}} - \Gamma$, 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 Inferencer

Unlike more traditional inverse identification procedures that rely either on gradient or gradient-free methods, the *Inferencer* class supports simulation-based inference that has been established as a powerful alternative approach.

The simulation-based inference is data-driven procedure supported by the sbi, `PyTorch`-based toolbox by Macke lab, [Tejero-Cantero2020].

In general, this method yields twofold improvement over point-estimate fitting procedures:

1. Simulation-based inference acts as if the actual statistical inference is performed, even in cases of extremly complex models with untractable likelihood function. Thus, instead of returning a single set of optimal parameters, it results in the approximated posterior distribution over unknown parameters. This is achieved by training a neural density estimator, details of which will be explained in depth later in the documentation.
2. Simulation-based inference uses prior system knowledge sparsely, using only the most important features to identify mechanistic models that are consistent with the recordings. This is achieved either by providing the predifend set of features, or by automatically extraciting summary features by using deep neural networks which is trained in parallel with neural density estimator.

The *Inferencer* class, in its core, is a fancy wrapper around the `sbi` package, where the focus is put on inferring the unknown parameters of the single-cell neuron models defined in Brian 2 simulator.

2.4.1 Neural density estimator

There are three main estimation techniques supported in `sbi` that the user can take the full control over seamlesly by using the *Inferencer*:

1. sequential neural posterior estimation (SNPE)
2. sequential neural likelihood estimation (SNLE)
3. sequential neural ratio estimator (SNRE)

2.4.2 Simulation-based inference workflow

The inferencer procedure is defined via three main steps:

1. step. Prior over unknown parameters needs to be defined, where the simplest choice would be uniform distribution given lower and upper bound (currently, this is only prior distribution supported through `brian2modelfitting` toolbox). After that, simulated data are generated given a mechanistic model with unknown parameters set as constants. Instead of taking the full output of the model, the neural network takes in summary data statistics of the output, e.g. instead of voltage trace as the output from a neuron model, we would feed a neural network with relevant electrophysiology features that outline the gist of the output sufficiently well.
2. step. A neural network learns association between the summary data statistics and unknown parameters (given the prior distribution over parameters). The learning method is heavily dependent on the choice of the inference technique.
3. step. The trained neural network is applied to the empirical data to infer posterior distribution over unknown parameters. Optionally, this process can be repeated by using the trained posterior distribution over parameters as the prior distribution proposal for a refined optimization.

2.4.3 Implementation

Go to *the tutorial section* for the in-depth implementation analysis.

2.4.4 References

2.5 Advanced Features

This part of documentation lists other features provided alongside or inside `Fitter` and `Inferencer` objects to allow users easier and a more flexible development when working on their own problems.

2.5.1 Parameters initialization

Whilst running *Fitter* or *Inferencer*, the user is able to pass the values of the parameters and variables that will be used as initial conditions when solving the differential equations defined in the neuron model.

Initial conditions should be passed by using an additional dictionary to the constructor:

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

```
fitter = TraceFitter(..., param_init = init_conds)
```

or

```
inferencer = Inferencer(..., param_init=init_conds)
```

2.5.2 Restart

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

By setting the `restart` argument in *fit()* to `True`, the user can restart the optimizer and the optimization will start from scratch.

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

The previously outlined `restart` argument is used in the similar fashion in *infer()* method. It is set to `False` by default, and each following re-call of the method will result in the multi-round inference. If the user wants amortized inference without using any knowledge from the previous round of optimization instead, the `restart` argument should be set to `True`.

2.5.3 Multi-objective optimization

In the case of *Fitter* classes, it is possible to fit more than one output variable at the same time by combining the errors for each variable. To do so, the user can specify several output variables during the initialization as follows:

```
fitter = TraceFitter(...,
                     output={'x': target_x,
                             'y': target_y})
```

If the fitter function uses a single metric, it is applied to both variables.

Note: This approach requires that the resulting error has the same units for all variables, i.e., it would not be possible to use the same *MSEMetric* on variables with different units, since the errors cannot be simply added up.

As a more general solution, the user can specify a metric for each variable and utilize their normalization arguments to make the units compatible (most commonly by turning both errors into dimensionless quantities). The normalization also defines the relative weights of all errors. For example, if the variable `x` has dimensions of mV and the variable `y` is dimensionless, the following metrics can be used to make an error of 10 mV in `x` to be weighed as much as an error of 0.1 in `y`

```
metrics = {'x': MSEMetric(normalization=10*mV),
           'y': MSEMetric(normalization=0.1)}
```

This has to be passed as the `metric` argument of the *`fit`* function.

In the case of the *`Inferencer`* class, switching from a single- to multi-objective optimization is seamless. The user has to provide multiple output variables during the initialization process the same way as for *`Fitter`* classes:

```
inferencer = Inferencer(...,
                        output={'x': target_x,
                                'y': target_y})
```

Later, during the inference process, the user has to define feautres for each output variable as follows:

```
posterior = inferencer.infer(...,
                             features={'x': list_of_features_for_x,
                                       'y': list_of_features_for_y})
```

If the user prefers automatic feature extraction, the `features` argument should not be defined (it should stay set to None).

Warning: If the user chooses to define a list of features for extracting the summary features, it is important to keep in mind that the total number of features will be increased as many times as there are output variables set for multi-objective optimization.

2.5.4 Callback function

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

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

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

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

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

...

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

2.5.5 OnlineTraceFitter

OnlineTraceFitter was created to work with long traces or large-scale optimization problems. This *Fitter* class uses online mean square error as a metric. When the *fit()* method is called there is no need of specifying a metric, which is by default set to None. The errors are instead calculated with `run_regularly` for each simulation.

```
fitter = OnlineTraceFitter(model=model,
                           input={'I': inp_traces},
                           output={'v': out_traces},
                           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])
```

2.5.6 Reference the target values in the equations

A model can refer to the target output values within the equations. For example, if the membrane potential trace v (i.e. `output_var='v'`) is used for the optimization, equations can refer to the target trace as `v_target`. This allows adding a coupling term such as: `coupling*(v_target - v)` to the equation that corresponds to state variable `v`, pulling the trajectory towards the correct solution.

2.5.7 Generate Traces

Fitter and *Inferencer* classes allow the user can to generate the traces with optimized parameters.

For a quick access to best fitted set of parameters *Fitter* classes provide ready to use functions:

- *generate_traces* inside *TraceFitter*;
- *generate_spikes* inside *SpikeFitter*.

These functions can be called after the fitting procedure is finalized 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()
```

On the other hand, since the *Inferencer* class is able to perform the inference of the unknown parameter distribution by utilizing output traces and spike trains simultaneously, `generate_traces` is used for both.

Once the approximated posterior distribution is built, the user is allowed to call `generate_traces` on *Inferencer* object. If only one output variable is used for the optimization of the parameters, the user does not have to specifiy output variable in the `generate_traces` method through `output_var` argument. If, for example, the multi-objective optimization is performed by using both output traces and spike trains and the user is interested in only times of spike events, `output_var` should be set to `'spike'`. Otherwise, if the user specifies a list of names or the `output_var` is not specified, a dictionary with keys set to output variable names and with their respective values, will be returned instead.

Customize the `generate` method for `Fitter`

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

```
fitter.generate(params=..., output_var=..., param_init=..., level=0)
```

where `params` should be a dictionary of parameters for which we generate the traces; `output_var` provides an option to pick one or more variables for visualization; with `param_init`, the user is able to define the initial values for differential equations in the model; and `level` allows for specification of the namespace level from which we are able to 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 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']
```

2.5.8 Results

Fitter classes store all of the parameters used by the optimizer as well as the corresponding errors. 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:

- `format='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},
 ...]
```

- format='dict':

```
{'g_na': array([176.4472297 , 212.57019659, ...]) * usiemens,
 'g_kd': array([ 43.82344525,  54.35309635, ...]) * usiemens,
 'gl': array([ 69.23559876, 134.68463669, ...]) * nsiemens,
 'errors': array([1.16788502, 0.5253008 , ...])}
```

- format='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
...
```

2.5.9 Posterior distribution analysis

Unlike *Fitter* classes, the *Inferencer* class does not keep track of all parameter values. Rather, it stores all training data for neural density estimator which will later be used for building the posterior distribution of each unknown parameter. Thus, the *Inferencer* does not returns best-fit values and corresponding errors, but the entire posterior distribution that can be used to draw samples from, compute descriptive statistics of parameters, analyize pairwise relationship between each to parameters, etc.

There are three methods that enable the comprehensive analysis of the posterior:

- *pairplot* - returns axes of drawn samples from the posterior in a 2-dimenstional grid with marginals and pairwise marginals. Using this method, the user is able to inspect the relationship for all combinations of distributions for each parameter;
- *conditional_pairplot* - visualizes the conditional pairplot;
- *conditional_corrcoeff* - returns the correlation matrix of a distribution conditioned with the user-specified condition.

To see this in action, go to our tutorial page and learn how to use each of these methods.

2.5.10 Standalone mode

Just like with regular Brian 2 scripts, all computations in the toolbox can be performed in `Runtime` mode (default) or `Standalone` mode. For details, please check the official Brian 2 documentation: https://brian2.readthedocs.io/en/stable/user/computation.html

To enable the `Standalone` mode, and to allowthe source code generation to C++ code, add the following code right after Brian 2 is imported, but before the simulation code:

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

Important notes:

Warning: In the `Standalone` mode, a single script should not contain multiple *Fitter* or *Inferencer* classes. Please, 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.

2.5.11 Embedding network for automatic feature extraction

If the `features` argument of the *`Inferencer`* class is not defined, automatic feature extraction from the given output traces will occur. By default, this is done by using the multi-layer perceptron that is trained in parallel with the neural density estimator of choice during the inference process. If the user wants to specify their own custom embedding network, it is possible to do so by creating a neural network by using `PyTorch` library and passing the instance of that neural network as an additional keyword argument as follows:

```
import torch
from torch import nn

...

class CustomEmbeddingNet(nn.Module):

    def __init__(self, in_features, out_features, ...):
        ...

    def forward(self, x):
        ...


in_features = out_traces.shape[1]
out_features = ...
embedding_net = CustomEmbeddingNet(in_features, out_features, ...)

...

inferencer = Inferencer(...)
inferencer.infer(...,
                 inference_kwargs={'embedding_net': embedding_net})
```

2.5.12 GPU usage for inference

It is possible to use the GPU for training the sdensity estimator. It is enough to specify the `sbi_device` to `'gpu'` or `'cuda'`. Otherwise, if not specified, or if set to `'cpu'`, training will be done by using the CPU.

Note: For default density estimators that are used either for SNPE, SNLE and SNRE, there are no significant speed-ups expected if the training is translocated to the GPU.

It is, however, possible to achieve a significant speed-up if the custom embedding network relies on convolutions to extract feautres. Such operations are known to achieve improvement in compuation time multifold.

2.6 Examples

2.6.1 Simple Examples

Following pieces of code show an example of two `Fitter` class calls and an `Inferencer` class call with possible inputs.

TraceFitter

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

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

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={'I': inp_traces},
                     output=out_spikes,
                     dt=0.1*ms,
                     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])
```

Inferencer

```
v_features = [
    lambda x: max(x[(t > t_start) & (t < t_end)]),  # AP max
    lambda x: mean(x[(t > t_start) & (t < t_end)]),  # AP mean
    lambda x: std(x[(t > t_start) & (t < t_end)]),  # AP std
    lambda x: mean(x[(t > .25 * t_start) & (t < .75 * t_start)]),  # resting
]
s_features = [lambda x: x.size]  # number of spikes in a spike train

inferencer = Inferencer(model=eqs, dt=0.1*ms,
                        input={'I': inp_traces},
                        output={'v': out_traces, 'spike': spike_times},
                        features={'v': v_features, 'spikes': s_features},
```

(continues on next page)

(continued from previous page)

```
                          method='exponential_euler',
                          threshold='m > 0.5',
                          refractory='m > 0.5',
                          param_init={'v': 'VT'})

posterior = inferencer.infer(n_samples=1_000,
                             n_rounds=2,
                             inference_method='SNPE',
                             gL=[20*nS, 40*nS],
                             C = [0.5*nF, 1.5*nF])
```

2.6.2 Multirun fitting 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
```

(continues on next page)

(continued from previous page)

```
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={'I': inp_traces*amp},
                     output={'v': out_traces*mV},
                     dt=dt, n_samples=20, 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()
```

2.6.3 Inference on Hodgin-Huxley model: simple interface

You can also download and run a similar example available here: `hh_sbi_simple_interface.py`

Here you can download the data: `input traces` `output traces`

```
from brian2 import *
from brian2modelfitting import *
import pandas as pd
```

To load the data, use the following:

```
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[[0, 1], 1:]
out_traces = df_out_traces.to_numpy()
out_traces = out_traces[[0, 1], 1:]
```

Then we have to define the model and its parameters:

```
area = 20_000*um**2
El = -65*mV
EK = -90*mV
ENa = 50*mV
VT = -63*mV
dt = 0.01*ms
eqs = '''
    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

    # free parameters
    g_na : siemens (constant)
    g_kd : siemens (constant)
    gl   : siemens (constant)
    Cm   : farad (constant)
'''
```

Let's also specify time domain for more convenient plotting afterwards:

```
t = arange(0, out_traces.shape[1]*dt/ms, dt/ms)
stim_start, stim_end = t[where(inp_traces[0, :] != 0)[0][[0, -1]]]
```

Now, we have to define features in order to create a summary statistics representation of the output data traces:

```
list_of_features = [
    lambda x: max(x[(t > stim_start) & (t < stim_end)]),  # max active potential
    lambda x: mean(x[(t > stim_start) & (t < stim_end)]),  # mean active potential
    lambda x: std(x[(t > stim_start) & (t < stim_end)]),  # std active potential
    lambda x: mean(x[(t > .25 * stim_start) & (t < .75 * stim_start)]),  # resting
]
```

We have to instantiate the object by using the class `Inferencer` in which the data and the list of features should be passed:

```
inferencer = Inferencer(dt=dt, model=eqs,
                        input={'I': inp_traces*amp},
                        output={'v': out_traces*mV},
                        features={'v': list_of_features},
                        method='exponential_euler',
                        threshold='m > 0.5',
                        refractory='m > 0.5',
                        param_init={'v': 'VT'})
```

Be sure that the names of parameters passed to the `infer` method correspond to the names of unknown parameters defined as constatns in the model equations.

```
posterior = inferencer.infer(n_samples=5_000,
                             n_rounds=3,
                             inference_method='SNPE',
                             density_estimator_model='mdn',
                             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*uF*cm**-2*area, 2*uF*cm**-2*area])
```

After the training of the neural density estimator stored accessible through `posterior` is done, we can draw samples from the approximated posterior distribution as follows:

```
samples = inferencer.sample((5_000, ))
```

In order to analyze the sampled data further, we can use the embedded `pairplot` method which visualizes the pairwise relationship between each two parameters:

```
limits = {'gl': [1e-9*siemens, 1e-07*siemens],
          'g_na': [2e-06*siemens, 2e-04*siemens],
          'g_kd': [6e-07*siemens, 6e-05*siemens],
          'Cm': [0.1*uF*cm**-2*area, 2*uF*cm**-2*area]}
labels = {'gl': r'$\overline{g}_{l}$',
          'g_na': r'$\overline{g}_{Na}$',
          'g_kd': r'$\overline{g}_{K}$',
          'Cm': r'$C_{m}$'}
inferencer.pairplot(limits=limits,
                    labels=labels,
                    ticks=limits,
                    figsize=(6, 6))
condition = inferencer.sample((1, ))
inferencer.conditional_pairplot(condition=condition,
                                limits=limits,
                                labels=labels,
                                ticks=limits,
                                figsize=(6, 6))
```

To obtain a simulated trace from a single sample of parameters drawn from posterior distribution, use the following code:

```
inf_traces = inferencer.generate_traces(output_var='v')
```

Let us now visualize the recordings and simulated traces:

```
inf_traces = inferencer.generate_traces(output_var='v')

nrows = 2
ncols = out_traces.shape[0]
fig, axs = subplots(nrows, ncols, sharex=True,
                    gridspec_kw={'height_ratios': [3, 1]}, figsize=(9, 3))
for idx in range(ncols):
    spike_idx = in1d(t, spike_times[idx]).nonzero()[0]
    spike_v = (out_traces[idx, :].min(), out_traces[idx, :].max())
    axs[0, idx].plot(t, out_traces[idx, :].T, 'C3-', lw=3, label='recordings')
    axs[0, idx].plot(t, inf_traces[idx, :].T/mV, 'k--', lw=2,
                     label='sampled traces')
    axs[1, idx].plot(t, inp_traces[idx, :].T/nA, lw=3, c='k', label='stimuli')
    axs[1, idx].set_xlabel('$t$, ms')
    if idx == 0:
        axs[0, idx].set_ylabel('$V$, mV')
        axs[1, idx].set_ylabel('$I$, nA')
handles, labels = [(h + l) for h, l
                   in zip(axs[0, idx].get_legend_handles_labels(),
                   axs[1, idx].get_legend_handles_labels())]
fig.legend(handles, labels)
tight_layout()
show()
```

2.6.4 Inference on Hodgin-Huxley model: flexible interface

You can also download and run this example by clicking here: `hh_sbi_simple_interface.py`

Here you can download the data: `input traces` `output traces`

```
from brian2 import *
from brian2modelfitting import *
import pandas as pd
```

To load the data, use the following:

```
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[[0, 1, 3], 1:]
out_traces = df_out_traces.to_numpy()
out_traces = out_traces[[0, 1, 3], 1:]
```

The model used for this example is the Hodgkin-Huxley neuron model. The parameters of the model are defined as follows:

```
area = 20_000*um**2
El = -65*mV
EK = -90*mV
ENa = 50*mV
VT = -63*mV
dt = 0.01*ms
eqs = '''
    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)/
```

(continues on next page)

(continued from previous page)

```
            (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

    # unknown parameters
    g_na : siemens (constant)
    g_kd : siemens (constant)
    gl   : siemens (constant)
    Cm   : farad (constant)
    '''
```

Now, let's define the time domain and start with the inferencer procedure manually:

```
t = arange(0, out_traces.shape[1]*dt/ms, dt/ms)
t_start, t_end = t[where(inp_traces[0, :] != 0)[0][[0, -1]]]

# Start with the regular instatiation of the class
inferencer = Inferencer(dt=dt, model=eqs,
                        input={'I': inp_traces*amp},
                        output={'v': out_traces*mV},
                        features={'v': [lambda x: max(x),
                                        lambda x: mean(x[(t > t_start) & (t < t_
↪end)]),
                                        lambda x: std(x[(t > t_start) & (t < t_
↪end)])]},
                        method='exponential_euler',
                        threshold='m > 0.5',
                        refractory='m > 0.5',
                        param_init={'v': 'VT'})
```

The prior should be initialized by defining the upper and lower bounds for each unknown parameter:

```
prior = inferencer.init_prior(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*uF*cm**-2*area, 2*uF*cm**-2*area])
```

If the input and output data for the training of the neural density estimator already exists, we can load it as follows:

```
path_to_data = ...
theta, x = inferencer.load_summary_statistics(path_to_data)
```

Otherwise, we have to generate training data and summary statistics from a given list of features:

```
theta = inferencer.generate_training_data(n_samples=10_000,
                                          prior=prior)
x = inferencer.extract_summary_statistics(theta)
```

And the data can be saved for the later use:

```
inferencer.save_summary_statistics(path_to_data, theta, x)
```

Finally, let's get our hands dirty and let's perform a single step of inference:

```
# amortized inference
inference = inferencer.init_inference(inference_method='SNPE',
                                      density_estimator_model='mdn',
                                      prior=prior)
# first round of inference where no observation data is set to posterior
posterior_amortized = inferencer.infer_step(proposal=prior,
                                            inference=inference,
                                            theta=theta, x=x)
```

After the posterior has been built, it can be stored as follows:

```
# storing the trained posterior without a default observation
path_to_posterior = ...
inferencer.save_posterior(path_to_posterior)
```

Now, as in the simple interface example, sampling can be performed via `sample` method where it is enough to define a number of parameters to be drawn from the posterior:

```
inferencer.sample((10_000, ))
```

Creating the pairwise relationship visualizations using the approximated posterior distribution

```
# define the label for each parameter
labels = {'gl': r'$\overline{g}_\mathrm{l}$',
          'g_na': r'$\overline{g}_\mathrm{Na}$',
          'g_kd': r'$\overline{g}_\mathrm{K}$',
          'Cm': r'$\overline{C}_{m}$'}
inferencer.pairplot(labels=labels)
```

It is possible to continue with the focused inference (to draw parameters from the posterior and to perform the training of a neural network to estimate the posterior distribution by focusing on a particular observation) by using a standard approach through `infer` method:

```
posterior_focused = inferencer.infer()
```

For every future call of `inferencer`, the last trained posterior will be used by default, e.g., when generating traces by using a single sample of parameters from a now non-amortized approximated posterior distribution:

```
inf_traces = inferencer.generate_traces()
nrows = 2
ncols = out_traces.shape[0]
fig, axs = subplots(nrows, ncols, sharex=True,
                    gridspec_kw={'height_ratios': [3, 1]},
                    figsize=(ncols * 3, 3))
for idx in range(ncols):
    axs[0, idx].plot(t, out_traces[idx, :].T, 'C3-', lw=3, label='recordings')
    axs[0, idx].plot(t, inf_traces[idx, :].T/mV, 'k--', lw=2,
                     label='sampled traces')
    axs[1, idx].plot(t, inp_traces[idx, :].T/nA, lw=3, c='k', label='stimuli')
   axs[1, idx].set_xlabel('$t$, ms')
    if idx == 0:
        axs[0, idx].set_ylabel('$V$, mV')
        axs[1, idx].set_ylabel('$I$, nA')
handles, labels = [(h + l) for h, l
                   in zip(axs[0, idx].get_legend_handles_labels(),
                   axs[1, idx].get_legend_handles_labels())]
fig.legend(handles, labels)
```

(continues on next page)

(continued from previous page)

```
tight_layout()
show()
```

CHAPTER 3

API reference

3.1 brian2modelfitting package

3.1.1 Subpackages and -modules

brian2modelfitting.fitter module

class brian2modelfitting.fitter.**Fitter**(*dt*, *model*, *input*, *output*, *n_samples*, *input_var=None*, *output_var=None*, *threshold=None*, *reset=None*, *refractory=None*, *method=None*, *param_init=None*, *penalty=None*, *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** (dic, ndarray or Quantity) – A dictionary given the input variable as the key and a 2D array of shape `(n_traces, time steps)` as the value, defining the input that will be fed into 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** (dict, Quantity or list) – Recorded output of the model that the model should reproduce. Should be given as a dictionary with the name of the variable as the key and the desired output as the value. The desired output has to be a 2D array of the same shape as the input when fitting traces with *TraceFitter*, or a list of spike times when fitting spike

trains with *SpikeFitter*. Can also be a list of several output 2D arrays or a single output array if combined with `output_var` (deprecated use).

- **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*. .. deprecated:: 0.5

  Use a dictionary for `input` instead.

- **output_var** (*str or list of str*) – The name of the output variable in the model or a list of output variables. Only needed when fitting traces with *TraceFitter*. .. deprecated:: 0.5

  Use a dictionary for `output` instead.

- **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_objective_errors

best_objective_errors_normalized

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*, or dict) – Child of Metric class, specifies optimization metric. In the case of multiple fitted output variables, can either be a single *Metric* that is applied to all variables, or a dictionary with a *Metric* for each variable.

- **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, additional_info)`. 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

n_neurons

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_objective_errors

best_objective_errors_normalized

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*, or dict) – Child of Metric class, specifies optimization metric. In the case of multiple fitted output variables, can either be a single *Metric* that is applied to all variables, or a dictionary with a *Metric* for each variable.
- **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, additional_info)`. 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

n_neurons

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_objective_errors

best_objective_errors_normalized

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*, or dict) – Child of Metric class, specifies optimization metric. In the case of multiple fitted output variables, can either be a single *Metric* that is applied to all variables, or a dictionary with a *Metric* for each variable.
- **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, additional_info)`. 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

n_neurons

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*, *output*, *dt*, *n_samples=60*, *input_var=None*, *output_var=None*, *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_objective_errors

best_objective_errors_normalized

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*, or dict) – Child of Metric class, specifies optimization metric. In the case of multiple fitted output variables, can either be a single *Metric* that is applied to all variables, or a dictionary with a *Metric* for each variable.
- **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, additional_info)`. 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

n_neurons

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*, *metric=None*, *callback='text'*, *calc_gradient=False*, *optimize=True*, *iteration=1000000000.0*, *level=0*, ***kwds*)

Refine the fitting results with a sequentially operating minimization algorithm. The `least_squares` algorithm from 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.
- **metric** (*`MSEMetric`* or dict, optional) – Optimization metrics to use. Since the refinement only supports mean-squared-error metrics, this is only useful to provide the `t_start`/`t_weights`/`normalization` values. In the case of multiple fitted output variables, can either be a single *`MSEMetric`* that is applied to all variables, or a dictionary with a *`MSEMetric`* for each variable. If not given, will reuse the metrics of a previous *`fit`* call.
- **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 `least_squares`.

Returns

- **parameters** (*dict*) – The parameters at the end of the optimization process as a dictionary.
- **result** (`scipy.optimize.OptimizeResult`) – The result of the optimization process.

Notes

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 which will internally calculate the sum of

squares.

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 appropriate size From list of dictionaries to dictionary of lists, with variables repeated for each trace

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.inferencer module

class brian2modelfitting.inferencer.**Inferencer**(*dt*, *model*, *input*, *output*, *features=None*, *method=None*, *threshold=None*, *reset=None*, *refractory=False*, *param_init=None*)
Bases: object

Class for a simulation-based inference.

It offers an interface similar to that of the *Fitter* class but instead of fitting, a neural density estimator is trained using a generative model which ultimately provides the posterior distribution over unknown free parameters.

To utilize simulation-based inference, this class uses a sbi library, for details see Tejero-Cantero 2020.

Parameters

- **dt** (*brian2.units.fundamentalunits.Quantity*) – Integration time step.
- **model** (*str or brian2.equations.equations.Equations*) – Single cell model equations.
- **input** (*dict*) – Input traces in the dictionary format where key corresponds to the name of the input variable as defined in model, and value corresponds to an array of input traces.
- **output** (*dict*) – Dictionary of recorded (or simulated) output data traces, where key corresponds to the name of the output variable as defined in model, and value corresponds to an array of recorded traces.
- **features** (*dict, optional*) – Dictionary of callables that take a 1-dimensional voltage trace or a spike train and output summary statistics. Keys correspond to output variable names, while values are lists of callables. If features are set to None, automatic feature extraction process will occur instead either by using the default multi-layer perceptron or by using the custom embedding network.
- **method** (*str, optional*) – Integration method.

- **threshold** (*str, optional*) – The condition which produces spikes. It should be a single line boolean expression.
- **reset** (*str, optional*) – The (possibly multi-line) string with the code to execute on reset.
- **refractory** (*bool or str, optional*) – Either the length of the refractory period (e.g., `2*ms`), a string expression that evaluates to the length of the refractory period after each spike, e.g., `'(1 + rand())*ms'`, or a string expression evaluating to a boolean value, given the condition under which the neuron stays refractory after a spike, e.g., `'v > -20*mV'`.
- **param_init** (*dict, optional*) – Dictionary of state variables to be initialized with respective values, i.e., initial conditions for the model.

References

- Tejero-Cantero, A., Boelts, J. et al. "sbi: A toolkit for simulation-based inference" Journal of Open Source Software (JOOS), 5(52):2505. 2020.

build_posterior(*inference*, ***posterior_kwargs*)

Return the updated inference and the neural posterior objects.

Parameters

- **inference** (*sbi.inference.NeuralInference*) – Instantiated inference object with stored paramaters and simulation outputs.
- **posterior_kwargs** (*dict, optional*) – Additional keyword arguments for `build_posterior` method in all `sbi.inference.NeuralInference`-type classes. For details, check the official `sbi` documentation.

Returns `sbi.inference.NeuralInference` object with stored paramaters and simulation outputs prepared for training and the neural posterior object.

Return type tuple

conditional_corrcoeff(*condition*, *density=None*, *limits=None*, *subset=None*, ***kwargs*)

Return the conditional correlation matrix of a distribution.

All but two parameters are conditioned with the condition as defined in the `condition` argument and the Pearson correlation coefficient is computed between the remaining two parameters under the distribution. This is performed for all pairs of parameters given whose names are defined in the `subset` argument. The conditional correlation matrix is stored in the 2-dimenstional array.

Check `sbi.analysis.conditional_density.conditional_corrcoeff` for more details.

Parameters

- **condition** (*numpy.ndarray*) – Condition that all but the one/two regarded parameters are fixed to.
- **density** (*neural posterior object, optional*) – Posterior probability density.
- **limits** (*dict, optional*) – Limits for each parameter. Keys correspond to parameter names as defined in the model, while values are lists with limits defined as the Brian 2 quantity objects. If None, min and max of the given samples will be used.
- **subset** (*list, optional*) – Parameters that are taken for conditional distribution, if None all parameters are considered.

- **kwargs** (*dict, optional*) – Additional keyword arguments for the `sbi.analysis.conditional_corrcoeff` function.

Returns Average conditional correlation matrix.

Return type numpy.ndarray

conditional_pairplot(*condition*, *density=None*, *points=None*, *limits=None*, *subset=None*, *labels=None*, *ticks=None*, ***kwargs*)

Plot conditional distribution given all other parameters.

The conditionals can be interpreted as slices through the density at a location given by condition.

Check `sbi.analysis.conditional_pairplot` for more details.

Parameters

- **condition** (*numpy.ndarray*) – Condition that all but the one/two regarded parameters are fixed to.
- **density** (*neural posterior object, optional*) – Posterior probability density.
- **points** (*dict, optional*) – Additional points to scatter, e.g., true parameter values, if known.
- **limits** (*dict, optional*) – Limits for each parameter. Keys correspond to parameter names as defined in the model, while values are lists with limits defined as the Brian 2 quantity objects. If None, min and max of the given samples will be used.
- **subset** (*list, optional*) – The names as strings of parameters to plot.
- **labels** (*dict, optional*) – Names for each parameter. Keys correspond to parameter names as defined in the model, while values are lists of strings.
- **ticks** (*dict, optional*) – Position of the ticks. Keys correspond to parameter names as defined in the model, while values are lists with ticks defined as the Brian 2 quantity objects. If None, default ticks positions will be used.
- **kwargs** (*dict, optional*) – Additional keyword arguments for the `sbi.analysis.conditional_pairplot` function.

Returns Figure and axis of conditional pairplot.

Return type tuple

extract_summary_statistics(*theta*, *level=0*)

Return the summary statistics for the process of training of the neural density estimator.

Parameters

- **theta** (*numpy.ndarray*) – Sampled prior with `n_samples` rows, and the number of columns corresponds to the number of free parameters.
- **level** (*int, optional*) – How far to go back to get the locals/globals.

Returns Summary statistics.

Return type numpy.ndarray

generate_traces(*n_samples=1*, *posterior=None*, *output_var=None*, *param_init=None*, *level=0*)

Generates traces for a single drawn sample from the trained posterior and all inputs.

Parameters

- **n_samples** (*int, optional*) – The number of parameters samples. If n_samples is larger than 1, the mean value of sampled set will be used.

- **posterior** (*neural posterior object, optional*) – Posterior distribution.
- **output_var** (*str or list*) – Name of the output variable to be monitored, it can also be a list of names to record multiple variables.
- **param_init** (*dict*) – Dictionary of initial values for the model.
- **level** (*int, optional*) – How far to go back to get the locals/globals.

Returns If a single output variable is observed, 2-dimensional array of traces generated by using a set of parameters sampled from the trained posterior distribution of shape (`self.n_traces`, number of time steps). Otherwise, a dictionary with keys set to names of output variables, and values to generated traces of respective output variables.

Return type brian2.units.fundamentalunits.Quantity or dict

generate_training_data (*n_samples*, *prior*)

Return sampled prior given the total number of samples.

n_samples [int] The number of samples.

prior [sbi.utils.BoxUniform] Uniformly distributed prior over given parameters.

Returns Sampled prior with the number of rows that corresponds to the `n_samples`, while the number of columns depends on the number of free parameters.

Return type numpy.ndarray

infer (*n_samples=None*, *theta=None*, *x=None*, *n_rounds=1*, *inference_method='SNPE'*, *density_estimator_model='maf'*, *inference_kwargs={}*, *train_kwargs={}*, *posterior_kwargs={}*, *restart=False*, *sbi_device='cpu'*, ***params*)

Return the trained posterior.

Note that if `theta` and `x` are not provided, `n_samples` has to be defined. Otherwise, if `n_samples` is provided, neither `theta` nor `x` are needed and will be ignored.

Parameters

- **n_samples** (*int, optional*) – The number of samples.
- **theta** (*numpy.ndarray, optional*) – Sampled prior.
- **x** (*numpy.ndarray, optional*) – Summary statistics.
- **n_rounds** (*int, optional*) – If `n_rounds` is set to 1, amortized inference will be performed. Otherwise, if `n_rounds` is integer larger than 1, multi-round inference will be performed. This is only valid if posterior has not been defined manually. Otherwise, if this method is called after posterior has already been built, multi-round inference is performed.
- **inference_method** (*str, optional*) – Inference method. Either SNPE, SNLE or SNRE.
- **density_estimator_model** (*str, optional*) – The type of density estimator to be created. Either `mdn`, `made`, `maf`, `nsf` for SNPE and SNLE, or `linear`, `mlp`, `resnet` for SNRE.
- **inference_kwargs** (*dict, optional*) – Additional keyword arguments for the *init_inference*.
- **train_kwargs** (*dict, optional*) – Additional keyword arguments for *train*.
- **posterior_kwargs** (*dict, optional*) – Additional keyword arguments for *build_posterior*.

- **restart** (*bool, optional*) – When the method is called for a second time, set to True if amortized inference should be performed. If False, multi-round inference with the existing posterior will be performed.
- **sbi_device** (*str, optional*) – Device on which the `sbi` will operate. By default this is set to `cpu` and it is advisable to remain so for most cases. In cases where the user provide custom embedding network through `inference_kwargs` argument, which will be trained more efficiently by using GPU, device should be set accordingly to either `gpu` or `cuda`.
- **params** (*dict*) – Bounds for each parameter. Keys should correspond to names of parameters as defined in the model equations, while values are lists with the lower and the upper bound with corresponding quantities of the parameter.

Returns Approximated posterior distribution over parameters.

Return type sbi.inference.posteriors.base_posterior.NeuralPosterior

infer_step(*proposal*, *inference*, *n_samples=None*, *theta=None*, *x=None*, *train_kwargs={}*, *posterior_kwargs={}*, **args*)
Return the trained neural density estimator.

Parameters

- **proposal** (`sbi.utils.torchutils.BoxUniform`) – Prior over parameters for the current round of inference.
- **inference** (`sbi.inference.NeuralInference`) – Inference object obtained via *init_inference* method.
- **n_samples** (*int, optional*) – The number of samples.
- **theta** (*numpy.ndarray, optional*) – Sampled prior.
- **x** (*numpy.ndarray, optional*) – Summary statistics.
- **train_kwargs** (*dict, optional*) – Additional keyword arguments for *train*.
- **posterior_kwargs** (*dict, optional*) – Additional keyword arguments for *build_posterior*.
- **args** (*list, optional*) – Additional arguments for *train*.

Returns Trained posterior.

Return type sbi.inference.posteriors.base_posterior.NeuralPosterior

init_inference(*inference_method*, *density_estimator_model*, *prior*, *sbi_device='cpu'*, ***inference_kwargs*)
Return instantiated inference object.

Parameters

- **inference_method** (*str*) – Inference method. Either SNPE, SNLE or SNRE.
- **density_estimator_model** (*str*) – The type of density estimator to be created. Either `mdn`, `made`, `maf`, `nsf` for SNPE and SNLE, or `linear`, `mlp`, `resnet` for SNRE.
- **prior** (*sbi.utils.BoxUniform*) – Uniformly distributed prior over free parameters.
- **sbi_device** (*str, optional*) – Device on which the `sbi` will operate. By default this is set to `cpu` and it is advisable to remain so for most cases. In cases where the user provides custom embedding network through `inference_kwargs` argument, which

will be trained more efficiently by using GPU, device should be set accordingly to either `gpu` or `cuda`.

- **inference_kwargs** (*dict, optional*) – Additional keyword arguments for different density estimator builder functions: `sbi.utils.get_nn_models.posterior_nn` for SNPE, `sbi.utils.get_nn_models.classifier_nn` for SNRE, and `sbi.utils.get_nn_models.likelihood_nn` for SNLE. For details check the official `sbi` documentation. A single highlighted keyword argument is a custom embedding network that serves a purpose to learn features from potentially high-dimensional simulation outputs. By default multi-layer perceptron is used if no custom embedding network is provided. For SNPE and SNLE, the user may pass an embedding network for simulation outputs through `embedding_net` argument, while for SNRE, the user may pass two embedding networks, one for parameters through `embedding_net_theta` argument, and the other for simulation outputs through `embedding_net_x` argument.

Returns Instantiated inference object.

Return type sbi.inference.NeuralInference

init_prior(***params*)

Return the prior uniform distribution over the parameters.

Parameters **params** (*dict*) – Dictionary with keys that correspond to parameter names, and the respective values are 2-element lists that hold the upper and the lower bound of a distribution.

Returns `sbi`-compatible object that contains a uniform prior distribution over a given set of parameters.

Return type sbi.utils.torchutils.BoxUniform

load_posterior(*f*)

Loads the density estimator state dictionary from a disk file.

Parameters **f** (*str or os.PathLike*) – Path to file either as string or `os.PathLike` object that contains file name.

Returns Loaded neural posterior with defined method family, density estimator state dictionary, the prior over parameters and the output shape of the simulator.

Return type sbi.inference.posteriors.base_posterior.NeuralPosterior

load_summary_statistics(*f*)

Load samples from a prior and the extracted summary statistics from a compressed `.npz` file.

Parameters **f** (*str or os.PathLike*) – Path to file either as string or `os.PathLike` object that contains file name.

Returns Sampled prior and the summary statistics arrays.

Return type tuple

n_neurons

Return the number of neurons that are used in `NeuronGroup` class while generating data for training the neural density estimator.

Unlike the *Fitter* class, *Inferencer* does not take the total number of samples directly in the constructor. Thus, this property becomes available only after the simulation is performed.

Parameters **None** –

Returns Total number of neurons.

Return type int

pairplot(*samples=None*, *points=None*, *limits=None*, *subset=None*, *labels=None*, *ticks=None*, ***kwargs*)

Plot samples in a 2-dimensional grid with marginals and pairwise marginals.

Check `sbi.analysis.plot.pairplot` for more details.

Parameters

- **samples** (*list or numpy.ndarray, optional*) – Samples used to build the pairplot.
- **points** (*dict, optional*) – Additional points to scatter, e.g., true parameter values, if known.
- **limits** (*dict, optional*) – Limits for each parameter. Keys correspond to parameter names as defined in the model, while values are lists with limits defined as the Brian 2 quantity objects. If None, min and max of the given samples will be used.
- **subset** (*list, optional*) – The names as strings of parameters to plot.
- **labels** (*dict, optional*) – Names for each parameter. Keys correspond to parameter names as defined in the model, while values are lists of strings.
- **ticks** (*dict, optional*) – Position of the ticks. Keys correspond to parameter names as defined in the model, while values are lists with ticks defined as the Brian 2 quantity objects. If None, default ticks positions will be used.
- **kwargs** (*dict, optional*) – Additional keyword arguments for the `sbi.analysis.pairplot` function.

Returns Figure and axis of the posterior distribution plot.

Return type tuple

sample(*shape*, *posterior=None*, ***kwargs*)

Return samples from posterior distribution.

Parameters

- **shape** (*tuple*) – Desired shape of samples that are drawn from posterior.
- **posterior** (*neural posterior object, optional*) – Posterior distribution.
- **kwargs** (*dict, optional*) – Additional keyword arguments for `sample` method of the neural posterior object.

Returns Samples taken from the posterior of the shape as given in `shape`.

Return type numpy.ndarray

save_posterior(*f*)

Save the density estimator state dictionary to a disk file.

Parameters

- **posterior** (*neural posterior object, optional*) – Posterior distribution over parameters.
- **f** (*str or os.PathLike*) – Path to file either as string or `os.PathLike` object that contains file name.

Returns

Return type None

save_summary_statistics(*f*, *theta=None*, *x=None*)
Save sampled prior and the extracted summary statistics into a single compressed `.npz` file.

Parameters

- **f** (*str or os.PathLike*) – Path to a file either as string or `os.PathLike` object that contains file name.
- **theta** (*numpy.ndarray, optional*) – Sampled prior.
- **x** (*numpy.ndarray, optional*) – Summary statistics.

Returns

Return type None

setup_simulator(*network_name*, *n_neurons*, *output_var*, *param_init*, *level=1*)
Return configured simulator.

Parameters

- **network_name** (*str*) – Network name.
- **n_neurons** (*int*) – Number of neurons which equals to the number of samples times the number of input/output traces.
- **output_var** (*str*) – Name of the output variable.
- **param_init** (*dict*) – Dictionary of state variables to be initialized with respective values.
- **level** (*int, optional*) – How far to go back to get the locals/globals.

Returns Configured simulator w.r.t. the available device.

Return type *brian2modelfitting.simulator.Simulator*

train(*inference*, *theta*, *x*, **args*, ***train_kwargs*)
Return the trained neural inference object.

Parameters

- **inference** (*sbi.inference.NeuralInference*) – Instantiated inference object with stored paramaters and simulation outputs prepared for the training process.
- **theta** (*torch.tensor*) – Sampled prior.
- **x** (*torch.tensor*) – Summary statistics.
- **args** (*tuple, optional*) – Contains a uniformly distributed proposal. Used only for SNPE, for SNLE and SNRE, proposal should not be passed to inference object, thus `args` should not be passed. The additional arguments should be passed only if the parameters were not sampled from the prior, e.g., during the multi-round inference. For SNLE and SNRE, this can be the number of round from which the data is stemmed from, e.g., 0 means from the prior. This is used only if the `discard_prior_samples` is set to True inside the `train_kwargs`.
- **train_kwargs** (*dict, optional*) – Additional keyword arguments for `train` method in the `sbi.inference.NeuralInference` class. The user is able to gain the full control over the training process by tuning hyperparameters, e.g., the batch size (by specifiying `training_batch_size` argument), the learning rate (`learning_rate`), the validation fraction (`validation_fraction`), the number of training epochs (`max_num_epochs`), etc. For details, check the official `sbi` documentation.

Returns Trained inference object.

Return type sbi.inference.NeuralInference

brian2modelfitting.inferencer.**calc_prior**(*param_names*, ***params*)
Return the prior distribution over given parameters.

Note that the only currently supported prior distribution is the multi-dimensional uniform distribution defined on a box.

Parameters

- **param_names** (*list*) – List containing parameter names.
- **params** (*dict*) – Dictionary with keys that correspond to parameter names, and the respective values are 2-element lists that hold the upper and the lower bound of a distribution.

Returns sbi-compatible object that contains a uniform prior distribution over a given set of parameters.

Return type sbi.utils.torchutils.BoxUniform

brian2modelfitting.inferencer.**configure_simulator**()
Return the configured simulator, which can be either *RuntimeSimulator*, object for the use with RuntimeDevice, or *CPPStandaloneSimulator*, object for the use with CPPStandaloneDevice.

Parameters **None** –

Returns Either *RuntimeSimulator* or *CPPStandaloneSimulator* depending on the currently active Device object describing the available computational engine.

Return type *brian2modelfitting.simulator.Simulator*

brian2modelfitting.inferencer.**get_full_namespace**(*additional_namespace*, *level=0*)
Return the namespace with added additional_namespace, in which references to external parameters or functions are stored.

Parameters

- **additional_namespace** (*dict*) – References to external parameters or functions, where key is the name and value is the value of the external parameter or function.
- **level** (*int, optional*) – How far to go back to get the locals/globals.

Returns Namespace with additional references to the external parameters or functions.

Return type dict

brian2modelfitting.inferencer.**get_param_dict**(*param_values*, *param_names*, *n_values*)
Return a dictionary compiled of parameter names and values.

Parameters

- **param_values** (*numpy.ndarray*) – Parameter values in a 2-dimensional array with the number of rows corresponding to a number of samples and the number of columns corresponding to a number of parameters.
- **param_names** (*list*) – List containing parameter names.
- **n_values** (*int*) – Total number of given values for a single parameter.

Returns Dictionary containing key-value pairs that correspond to a parameter name and value(s).

Return type dict

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

revert_normalization(*error*)

Revert the normalization to recover the error before normalization.

Parameters **error** (*Quantity or float*) – The normalized error.

Returns **raw_error** – The error before normalization

Return type Quantity or float

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`

revert_normalization(*error*)
Revert the normalization to recover the error before normalization.

Parameters **error** (*Quantity or float*) – The normalized error.

Returns **raw_error** – The error before normalization

Return type Quantity or float

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

revert_normalization(*error*)
Revert the normalization to recover the error before normalization.

Parameters **error** (*Quantity or float*) – The normalized error.

Returns **raw_error** – The error before normalization

Return type Quantity or float

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`

revert_normalization(*error*)
Revert the normalization to recover the error before normalization.

Parameters **error** (*Quantity or float*) – The normalized error.

Returns **raw_error** – The error before normalization

Return type Quantity or float

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

revert_normalization(*error*)
Revert the normalization to recover the error before normalization.

Parameters **error** (*Quantity or float*) – The normalized error.

Returns **raw_error** – The error before normalization

Return type Quantity or float

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

revert_normalization(*error*)
Revert the normalization to recover the error before normalization.

Parameters **error** (*Quantity or float*) – The normalized error.

Returns **raw_error** – The error before normalization

Return type Quantity or float

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 $1 + 2\frac{|r_{\mathrm{data}} - r_{\mathrm{model}}|}{r_{\mathrm{data}}} - \Gamma$ (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.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 or callable, optional) – The optimization method. By default differential evolution, can be chosen by name from any method in Nevergrad registry. Alternatively, a callable object can be provided.
- **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*, *rounds*, ***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
- **rounds** (*int*) – Number of rounds (for budget calculation)
- ****params** – bounds for each parameter

Returns **popsize** – The actual population size that will be used by the algorithm. Does not always correspond to popsize, since some algorithms have minimal/maximal population sizes.

Return type int

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*, *rounds*, ***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
- **rounds** (*int*) – Number of rounds (for budget calculation)
- ****params** – bounds for each parameter

Returns **popsize** – The actual population size that will be used by the algorithm. Does not always correspond to `popsize`, since some algorithms have minimal/maximal population sizes.

Return type int

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*, *rounds*, ***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
- **rounds** (*int*) – Number of rounds (for budget calculation)
- ****params** – bounds for each parameter

Returns **popsize** – The actual population size that will be used by the algorithm. Does not always correspond to popsize, since some algorithms have minimal/maximal population sizes.

Return type int

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.tests package

brian2modelfitting.tests.**run**()

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*, *additional_info*)
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*, *additional_info*)
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

Bibliography

[Hodgkin1952] Hodgkin, A. L., and Huxley, A. F. "A quantitative description of membrane current and its application to conduction and excitation in nerve" Journal of Physiology 117(4):500-44. 1952. doi: 10.1113/jphysiol.1952.sp004764

[Greenberg2019] Greenberg, D. S., Nonnenmacher, M. et al. "Automatic posterior transformation for likelihood-free inference" 36th International Conference on Machine Learning (ICML 2019). 2019. Available online: arXiv:1905.07488

[Rodrigues2020] Rodrigues, P. L. C. and Gramfort, A. "Learning summary features of time series for likelihood free inference" 3rd Workshop on Machine Learning and the Physical Sciences (NeurIPS 2020). 2020. Available online: arXiv:2012.02807

[Tejero-Cantero2020] Tejero-Cantero, A., Boelts, J. et al. "sbi: A toolkit for simulation-based inference" Journal of Open Source Software 5(52):2505. 2020. doi: 10.21105/joss.02505

Python Module Index

b

Index

E

F

G

I

L

M

N

O

P

R

S

T